© 2023 Jiangran Wang

CHURN-TOLERANT LEADER ELECTION ON WEAKLY-CONSISTENT MEMBERSHIP

BY

JIANGRAN WANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Professor Indranil Gupta

ABSTRACT

Many distributed systems suffer from churn, such as peer-to-peer file-sharing networks and distributed database systems. Churn in distributed systems is the frequent joining and leaving of nodes, leading to instability and challenges in coordination. Distributed applications running in such systems require underneath them a membership protocol that updates (local) membership lists in spite of churn. On top of this membership layer run coordination protocols including (but not limited to) leader election, mutual exclusion, agreement, etc. In this thesis, we present: i) a membership protocol called Medley-F; ii) a family of leader election protocols that are churn-tolerant; and iii) two protocols that estimate churn in distributed systems.

First, we present Medley-F, a weakly-consistent membership protocol for IoT networks. It is an enhanced version of an existing system (Medley), and Medley-F utilizes active and passive strategies to reduce failure detection tail latency. We show both simulation results, as well as deployment results, atop Raspberry Pi devices.

Second, we present a family of four leader election protocols that are churn-tolerant (or c-tolerant). The key ideas are to: i) involve the minimum number of nodes necessary to achieve safety; ii) use optimism so that decisions are made faster when churn is low; iii) incorporate a preference for electing healthier nodes as leaders. We present experimental results from both a trace-driven simulation and our implementation atop Raspberry Pi devices, including a comparison against Zookeeper.

Third, we present two protocols to dynamically estimate churn with the underlying membership changes continuously. The key ideas are to: i) sample memberships from a small subset of nodes; ii) utilize intermediate results during leader election. We show simulation results of the estimation accuracy of both protocols and the effect of underestimating churn on leader election safety.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Indranil Gupta for his invaluable advice and guidance. I would also like to thank my colleagues and collaborators, Rui Yang, Xiaojuan Ma, and Anna Karanika, for their continuous support and invaluable help.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Contributions of this Thesis 3
1.2 Thesis Organization 3

CHAPTER 2 BACKGROUND 5
2.1 Failure Detector 5
2.2 Membership Protocol 6
2.3 Leader Election 7
2.4 Related Work 8
2.5 System Model 9

CHAPTER 3 MEDLEY-F MEMBERSHIP SERVICE 11
3.1 Motivation 11
3.2 Medley-F: Feedback-Based Target Selection 12

CHAPTER 4 MEDLEY-F EXPERIMENTS 15
4.1 Simulation 15
4.2 Deployment 21

CHAPTER 5 CHURN-TOLERANT LEADER ELECTION 24
5.1 Motivation 24
5.2 Protocol Design 25
5.3 Formal Analysis 34

CHAPTER 6 LEADER ELECTION EXPERIMENTS 38
6.1 Trace-Driven Simulation 38
6.2 Deployment with Raspberry Pis 46

CHAPTER 7 ESTIMATING CHURN PARAMETER c 51
7.1 Motivation 51
7.2 Protocol Design 52
7.3 Simulation 56
7.4 Conclusion 61

CHAPTER 8 CONCLUSION 62
8.1 Summary 62
8.2 Future Work 63

REFERENCES 65

CHAPTER 1

INTRODUCTION

Driven by the promise of autonomous operation, self-adaptability, low latency, and bandwidth [1], the global edge computing market size is expected to reach 155.9 billion USD by 2030 [2], with over 29 billion IoT devices by 2030 [3]. Edge computing scenarios range from "stable" ones like smart homes, Industry 4.0, and infrastructure deployments (e.g., smart bridges, roads, etc.), to "dynamic" settings—examples include robots in remote or inhospitable terrains (e.g., emergency rescue and recovery scenarios, battlefields, smart farms) [4, 5, 6], to constellations of satellites (low earth LEO, or medium earth MEO) [7, 8], to fast-moving vehicle platoons [9]. While a bulk of research today deals with the former stable edge settings, the rapid recent growth of the latter dynamic settings brings forth many problems that remain unsolved.

Coordination tasks—such as consensus protocols, distributed transactions, and distributed task scheduling—are widely used in real-world applications. For example, [10] analyzes the performance of (PBFT) consensus for blockchain networks, Java Transaction API (JTA) [11] is a standard Java interface that manages distributed transactions, and Apache Spark utilizes distributed task scheduling for machine learning and big data processing [12]. Such tasks often require knowledge of membership of the system: in consensus, the system needs to know the value at each node to ensure everyone reaches an agreement; in distributed task scheduling, we need to know the availability and resource capability at each node to schedule tasks efficiently.

This thesis makes three contributions applicable to edge and IoT networks that suffer from churn. The three contributions are: i) Medley-F, a fully-decentralized membership protocol for IoT networks; ii) a family of churn-tolerant leader election protocols; iii) two protocols that dynamically estimate churn in a given system.

The first contribution of this thesis is Medley-F [13], a fully-decentralized

membership protocol for IoT networks. Its main goal is to detect device failures (crashes) and update membership lists at non-faulty nodes. At large scale, failures are the norm rather than the exception. In distributed systems, it is important for other devices to be aware of any device failures and take corrective action, sometimes even informing the user. Membership is an important building block atop which many other distributed protocols are built. However, current IoT methods for detecting failures are usually centralized or semi-centralized [14, 15, 16, 17], relying on a central clearinghouse that stores information about active nodes. These methods often require access to a cloud, which may not always be practical or feasible.

The second contribution [18] of this thesis is layered atop Medley-F, and it tackles the classical problem of leader election. A leader election protocol can be initiated by any one node, and aims to satisfy both: 1) (Safety) elect a unique leader that has the single "best attribute" (e.g., the lowest hash ID, or the most amount of data, etc.), and 2) (Liveness) inform all non-faulty nodes of this single unique leader's ID. Leader election is a key building block for coordination among nodes in dynamic edge settings, especially when remoteness means no central hub or infrastructure is present nearby to perform coordination actions. For instance, robots in rescue and recovery scenarios, or smart farms [6], need a leader for coordination when no human operator is nearby. A second example—LEO or MEO satellites [7]—move very fast (finishing an orbit in under 2 hrs), and so the set of satellites above a geographical area (e.g., a state) is a dynamic group and yet needs a leader satellite for coordination actions (since ground stations are sparse and may be absent in that area). A third example is vehicle platoons using a leader to achieve autonomous caravan driving [9].

The third contribution of this thesis is churn estimation for distributed systems. Churn is a common phenomenon in distributed systems, but the exact value is often difficult to estimate accurately, as the nodes constantly join and leave the system. The churn rate in distributed systems and peer-to-peer networks is well-studied [19, 20, 21]. Our work dynamically estimates churn with the underlying membership changing continuously.

1.1 Contributions of this Thesis

This thesis makes the following contributions:

- We present an enhanced version of a previous failure detector named Medley [22], which we call Medley-F [13]. Compared to Medley, Medley-F reduces tail latency in failure detection with unlucky nodes by utilizing Active-feedback strategy (where node monitors whether itself is unlucky) and Passive-feedback strategy (where node monitors whether neighboring nodes are unlucky)
- We propose a family progression of four variants for churn-tolerant (or *c-tolerant*) election [18] that (respectively) involve the fewest messages, have *optimism* by completing quickly in the common (consistent) case, accommodate a "health" *preference* for the leader, and include a hybrid optimistic-preference variant.
- We prove Safety and Liveness of all our protocols and analyze the optimality of their message complexity.
- We present both simulation and Raspberry Pi deployment results of our churn-tolerant election protocols, where we inject traces from the Medley-F membership protocol. We also show that our protocols consume far fewer resources than stock solutions like Zookeeper [23].
- We present two protocols for estimating churn. Obtaining the optimal value of churn is essential for our churn-tolerant leader election protocols to run accurately and efficiently. We also show simulations of our churn estimation protocols and the effect of churn underestimation.

1.2 Thesis Organization

This thesis is organized as follows. Chapter 2 presents background, related works, and system model. Chapter 3 presents the design of Medley-F, a weakly-consistent membership service, and Chapter 4 shows experiment results. Chapter 5 presents the design of a family of four churn-tolerant leader election protocols and their formal analysis. Chapter 6 presents both simulation and Raspberry Pi deployment results of our election protocols. Chapter 7 presents two different protocols on estimating churn in a system and

their corresponding simulations. Chapter 8 summarizes this thesis and discusses potential future works.

CHAPTER 2

BACKGROUND

In this chapter, we discuss some preliminary concepts related to the thesis contributions. Sec. 2.1 talks about failure detectors, Sec. 2.2 talks about different types of membership protocols, and Sec. 2.3 talks about leader election. Sec. 2.4 presents related works, and Sec. 2.5 presents our system model.

2.1 Failure Detector

In large-scale systems, failures are the norm rather than the exception. When a device fails, other affected devices need to know about it and take appropriate corrective action, and, in some cases, inform the human user. This is a very common way of building internet-based and datacenter-based distributed systems. In the IoT environment, examples of corrective actions after failure include (but are not limited to): backup actions to ensure user needs are met (e.g., maintaining sufficient lighting in an area), re-initiating and re-replicating device schedules that were stored on failed devices (e.g., timed schedules), informing the upper management layer, informing the user, etc.

Failure detector protocols for internet-based distributed systems fall into two categories: heartbeat-based (or lease-based) and ping-based. Heartbeat-based protocols [24, 25, 26] have each node send periodic heartbeats to one or more other monitor nodes; when a node n_i dies, its heartbeats stop, the monitors time out, and detect the node n_i as failed. Ping-based protocols [27, 28] have each node periodically ping randomly selected target nodes from the system. Analysis in [27] has shown that compared to heartbeat protocols, ping-based protocols are faster at detecting failures and impose less network traffic, and can completely detect failures.

2.2 Membership Protocol

There are two major kinds of membership protocols in asynchronous distributed systems—strong and weak.

Strongly-Consistent Membership: The strictest variants assume that membership lists at each alive node are identical at all times. This implies that membership updates are delivered to all nodes at the exact same time, which is impossible to achieve in asynchronous systems [29, 30]. A slightly relaxed variant (within strong membership) assumes that membership updates will reach each node in the same order, but possibly at different physical arrival times. Examples include virtual synchrony [31], Raft [32], Zookeeper [23], and others [33, 34]. Yet the overhead of strongly-consistent membership protocols scales poorly with churn, rendering nodes unable to do useful work. For instance, this is why a typical Zookeeper cluster only contains between 3 and 7 servers.

Weakly-Consistent Membership: A weakly-consistent membership propagates membership updates to all nodes eventually, but it does not guarantee either ordering or timing of the updates. Examples include gossip-style heartbeating [25], SWIM [28], and other IoT membership protocols [13]. These protocols scale better than strongly-consistent membership protocols and have been used in systems with thousands of participating nodes [35, 36]. However, membership inconsistencies create a mismatch with the desire to provide a strong safety property above it. Concretely, a node M_i could be missing in other nodes' membership lists if M_i just joined or it is mistakenly detected as failed (false positive) by some other nodes. A node M_i could also falsely exist at another node's membership list, but M_i is actually failed, as failure detection is not instantaneous.

Membership in Edge Settings: Edge settings are better suited to weakly-consistent membership protocols. Strongly-consistent membership protocols have difficulty scaling in asynchronous systems, as they incur high bandwidth and tend to splinter the node group beyond a few 10s of nodes [37, 31, 29, 38]. Weakly-consistent membership protocols [39, 19, 13] only guarantee eventual delivery of membership changes without order, but they scale well and use low bandwidth. Hence, all election protocols presented in this paper are layered over a weakly-consistent membership.

2.3 Leader Election

Leader election is a critical process in distributed systems that involves selecting a leader among a group of peers. In a distributed system, a leader is responsible for coordinating and managing the tasks performed by other nodes in the system. The leader is typically the node with the most up-to-date information and the highest level of availability. It ensures that there is always a designated node responsible for coordinating and managing the system.

The leader election process is important because it helps to ensure that the distributed system is reliable, efficient, and fault-tolerant. In the absence of a leader, nodes may operate independently and may not be able to coordinate with each other effectively. This can result in inconsistencies, conflicts, and failures within the system.

One of the commonly used leader election protocols is the Bully Algorithm, which involves the nodes communicating with each other to determine which node has the highest priority. The node with the highest priority is elected as the leader, and all other nodes recognize this node as the leader. Another algorithm is the Ring Algorithm, where each node is assigned a unique identifier and the nodes form a logical ring. The nodes then pass messages around the ring until a node is identified as the leader.

An election run must satisfy the following two properties:

Definition 1. ***Liveness:*** *Protocol terminates, i.e., each alive node eventually elects a leader (sets its local leader variable to a non-null value).*

Definition 2. ***Safety:*** *At the end of the leader election, each alive node only sets its local leader variable to that unique non-faulty node which has the lowest hash of all non-faulty nodes currently in the system.*

We assume the election is initiated by a single "initiator" node. This node may be the one that detects the failure of the old leader or is the bootstrap node when the system boots up. We do not tackle initiator failure because weak membership protocol failures are eventually detected (so *some* node will eventually detect the old leader's failure). For ease of exposition, our description of the protocols assumes a single initiator. To handle multiple initiators—any initiator that hears of a lower ID initiator does not complete its protocol; thus only the lowest ID initiator completes.

2.4 Related Work

2.4.1 Related Work in Failure Detection

Classical Failure Detection: Failure detection in data-centers is well-studied. The earliest failure detectors send periodic "I am alive" heartbeats [24] to all other or a subset of nodes. Timeout on the next heartbeat leads to failure detection. Heartbeats may be multicast or gossiped [25] or spread hierarchically [40]. As described earlier, SWIM [28] is the inverse of heartbeating, relying on pinging, and has bandwidth provably within a constant factor of optimal. FUSE [41] disseminates failure information via applications, to reduce network costs.

Failure Detection in IoT Networks: Existing IoT failure detection schemes largely focus on data anomalies. Sympathy [42] uses flooding and aggregates distributed data at the sink, detecting failure by finding insufficient flow of incoming data. Memento [43] uses a tree for failure monitoring, limiting its scalability under failures. Network-level delays and packet traces can be used for failure detection [44, 45]. Yet, these are hard to analyze mathematically. DICE [17] uses context (e.g., sensor correlation, state transition probabilities) to identify anomalous readings and their sensor nodes. Asim et al. [46] partitions the network into cells, detects failures within cells, and multicasts it across cells–this however assumes a homogeneous network.

2.4.2 Related Work in Churn-tolerant Leader Election

Broadly, the topic of layering consistent services over inconsistent substrates has received recent attention. [47] implements transactional systems on top of inconsistent replication in distributed systems, and [38] supports virtual synchrony over gossiping. The key idea in [47] and [38] are similar: they both *reactively* try to fix inconsistency issues raised from lower layers, as opposed to our (membership-based) approach which is *proactive*. [48] builds a microservices OS over inconsistent networks by integrating strong consistency properties into the membership layer itself. Neither of the aforementioned papers builds over inconsistent membership lists, nor did they propose fully-distributed election protocols. Some work [49, 50] provides *probabilistic* safety property for distributed protocols atop weak membership, while

we provide 100% safety. Ad-hoc routing [36] uses gossip [37, 51] to spread information fast, but requires flooding to maintain safety. To the best of our knowledge, our work is the first to explore the challenges and mismatches of layering strongly-consistent distributed protocols directly over weak membership layers.

Churn is a common phenomenon in peer-to-peer systems and past work has built distributed hash tables and applications that are churn-tolerant [52, 53], yet the notions of safety provided therein are merely probabilistic or best-effort (while we provably guarantee safety). Classical leader election solutions such as the Bully Algorithm [54] or consensus-based election protocols like Zookeeper [23], Paxos [55], and Raft [32], assume full and correct membership. Election in mobile ad-hoc networks that are subject to partitions exists [56, 57], yet they assume FIFO links and consistency within each partition (these partition-tolerant techniques can be applied orthogonally to us as we do not consider partitions). Self-stabilizing leader election [58, 59, 60] aims to recover from transient faults, but they assume that the underlying "overlay" (ring, tree, etc.) stabilizes after failures. We assume no membership convergence, and our membership lists can be inconsistent all the time.

2.5 System Model

For each of our contributions, we assume a full-asynchronous system where messages have arbitrary delay but are delivered eventually, and nodes may crash by failing. Like classical literature, we assume a known upper-bound f on the number of simultaneously failed nodes (after the protocol quiesces, further f nodes may fail).

For our churn-tolerant leader election protocols and churn estimation, we assume a weakly-consistent membership protocol [25, 28, 13] runs in the background at each node and continually updates its local membership. Each node in the system can join, leave, or fail (crash) arbitrarily, and the membership protocol only provides the property that such updates are eventually propagated to non-faulty nodes. We also assume multicast is eventually reliable, e.g., via gossip [61] or R-multicast [62]. To summarize: in the rest of this paper, a *churned setting* means a weakly-consistent membership and

eventually reliable multicast running at each node.

CHAPTER 3

MEDLEY-F MEMBERSHIP SERVICE

In this chapter, we present a weakly-consistent membership service for IoT settings, Medley-F, which is an enhanced version of Medley. Medley-F utilizes two strategies—active and passive—to reduce the failure detection tail time of Medley. Sec. 3.1 summarizes the main idea of Medley and discusses the motivation of introducing Medley-F. Sec. 3.2 presents the algorithm design of the two strategies.

3.1 Motivation

Medley [22, 13] is a fully-distributed failure detector for IoT deployments running over a wireless ad-hoc network. It is the first fully-decentralized membership service for IoT distributed systems, which maintains a dynamic membership list containing a list of currently alive nodes in the system at each IoT node. This allows fully distributed applications to be built atop Medley.

The base Medley system [22] proposes a new spatial ping-target selection strategy that prefers nearer nodes but also has some probability of pinging farther nodes. A node chooses to ping a given target with probability proportional to $\frac{1}{r^m}$, where r is the distance to the target and m is a fixed exponent. The hybrid of the uniform-random and the always-local target selection utilizes a mix of nearer and farther ping targets to gain the advantages of both approaches. The proposed spatial ping-target selection strategy reduces network traffic, avoids congestion, and provides load balancing. Medley also proposes a time-bounded design to restrict the upper bound of failure detection time.

One of the key issues in the base Medley system is the long detection time of unlucky nodes in the system. These unlucky nodes are affected by uneven

distribution and the network is not aware of their presence. The longer Medley tries to reduce communication costs, the more difficult it becomes to identify these problematic nodes, resulting in a longer delay.

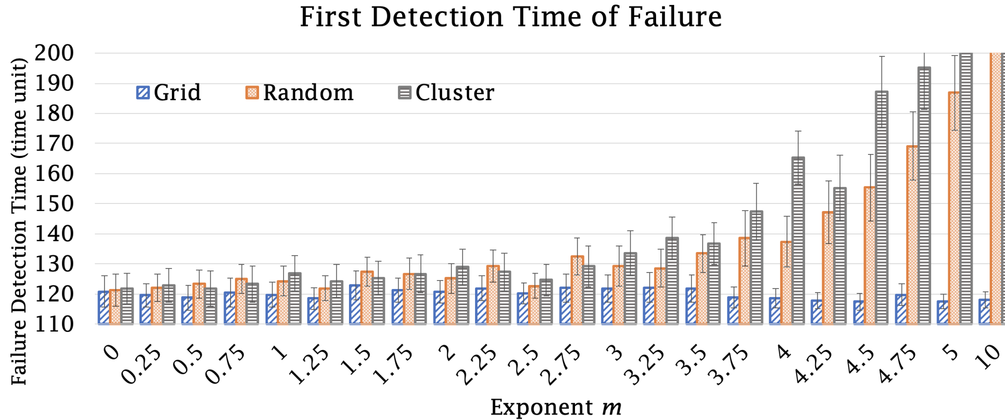


Figure 3.1: *Medley first detection time of failure with different exponent m*

Fig. 3.1 shows the first detection time of failure for Medley with different exponent m and different topology. As m increases, the failure detection time increases exponentially for random and cluster topology. These come from the long tail of the detection time of unlucky nodes. In Medley-F [13], we aim to reduce the detection time of unlucky nodes while still utilizing the benefit of low communication cost under high exponent m.

3.2 Medley-F: Feedback-Based Target Selection

Medley, as described so far, may have a long tail of detection time for a small subset of nodes. We define an *unlucky node* n_i as one whose neighbors have all their (respective) neighbors much closer to themselves, while n_i is relatively far from each of its neighbors. When the exponent m is high and pings stay local, unlucky nodes have fewer pingers. If an unlucky node fails, its detection time will be longer than other nodes. To reduce this tail, we explore a variant of Medley, called Medley-F. Medley-F consists of competing approaches: an *active approach* wherein a node actively realizes it is unlucky and a *passive approach* wherein other nodes realize the unlucky node. In both cases, the modified Medley adjusts the rate of pinging to the unlucky node—permanently for the active approach and temporarily for the passive approach.

3.2.1 Active Feedback Strategy

In active-feedback, every node *actively* monitors itself and reports its unluckiness to its 1-hop neighbors. These neighbors adjust their pinging probability to the unlucky node.

Member Self Monitoring Each node estimates, via exponential averaging, the average interval of *incoming pings*. Given a new measurement M of pinging interval, Medley-F updates the estimate pinging interval I via exponential averaging: $I \leftarrow (1-\alpha) \cdot I + \alpha \cdot M$. We use $\alpha = 0.125$ in our implementation. If the estimate I is above `ACTIVE_TIMEOUT`, M_i considers itself as `UNLUCKY` and reports to all its direct neighbors. We recommend setting `ACTIVE_TIMEOUT` to be less than suspicion timeout (e.g. half of suspicion timeout), so that an unlucky node can report its unluckiness and potentially update its aliveness to other nodes in a timely manner.

Unlucky handling When a node M_j receives an `UNLUCKY` report from a 1-hop neighbor M_i, M_j uses Algo. 1 to boost the pinging probability of M_i to reach the average pinging rate for other non-unlucky (or lucky) nodes.

We describe Algo. 1 via an example. Consider a node M_5 is maintaining pinging probabilities of [0.43, 0.3, 0.17, 0.1] for its four neighbors M_1 - M_4. Say M_5 receives an `UNLUCKY` report from M_4 ($p = 0.1$). Following Lines 1 - 3, the target (average) pinging probability from M_5 to M_4 will be $(0.43+0.3+0.17+0.1)/4 = 0.25$. Because M_1 and M_2 have above average probabilities, they become *probability sponsors* (Line 5). The excess probability of $0.18 + 0.05 = 0.23$ is provided to boost M_4 from 0.1 to 0.25. Since $0.23 > 0.15$, M_1 and M_2 respectively and proportionally provide $0.18 \times \frac{0.15}{0.23} = 0.12$ and $0.05 \times \frac{0.15}{0.23} = 0.03$. The final probability list at M_5 is $[0.31, 0.27, 0.17, 0.25]$.

This approach boosts unlucky nodes and reduces pinging only to nodes with already-high ping rates.

3.2.2 Passive Feedback Strategy

The passive-feedback strategy is the reverse of active-feedback, and uses neighbors to detect an unlucky node. In passive-feedback, each node M_i actively maintains timestamp information about the *last contact* from other

Algorithm 1 Probability modification for a single node M_j after receiving `UNLUCKY` reports of M_i

Require: M_j's Runtime Probability List $\mathcal{P}_{M_j}$

1: ▷ Get target probability for unlucky node M_i
2: $\mathcal{P}_{above}$ = all p_k in $\mathcal{P}_{M_j}$ with $p_k > p_{M_i}$ or $k = M_i$
3: p_{target} = mean($\mathcal{P}_{above}$)

4: ▷ Migrate probability from high-prob nodes to M_i
5: $\mathcal{P}_{sponsor}$ = all (p_k - p_{target}) in $\mathcal{P}_{M_j}$ with $p_k > p_{target}$
6: **for** each p_k in $\mathcal{P}_{sponsor}$ **do**
7: $\quad p_k$ -= $(p_k - p_{target}) \times \frac{p_{target} - p_{M_i}}{\text{sum}(\mathcal{P}_{sponsor})}$
8: **end for**
9: $p_{M_i} = p_{target}$

members. Such contact may be either through a direct contact where 1-hop neighbor sends or forwards a message, or via an indirect contact where a multi-hop member originates a message. To reduce the message payload, we do not keep information for intermediate routing path nodes. When selecting the next ping target, M_i flips a coin with probability $p_{passive}$, and if it turns up heads, M_i does a *passive check*. During a passive check, M_i looks at its membership list and checks whether any node has not contacted M_i in the last `PASSIVE_TIMEOUT` time units. If any, M_i suspects it as unlucky and randomly selects one of them as the next ping target. In our implementation we set to a less aggressive $p_{passive} = 0.1$.

In active-feedback, the unlucky members that a node gets notified about are always 1-hop neighbors, while in passive-feedback the reported unlucky nodes are often "far" members whose information tends to stay local (at high m). While pinging such far nodes involves more hop-to-hop communication, passive-feedback can: i) still save considerable bandwidth compared to basic SWIM since the majority of ping targets are still local, and ii) avoid extra messages to report unlucky nodes that active-feedback needs.

CHAPTER 4

MEDLEY-F EXPERIMENTS

In this chapter, we present the experiment results of Medley-F. The results include the comparison between base Medley, Active-feedback, Passive-feedback, and both strategies combined. Sec. 4.1 shows the simulation results, and Sec. 4.2 shows the Raspberry Pi deployment results.

4.1 Simulation

4.1.1 Simulation Results

We developed a *second matching* simulator, in Java, that *uses the same code as our Raspberry Pi deployment* but without NS3's fine-grained link layer modeling. We evaluated Medley-F in two topologies: i) Random (nodes are randomly placed), and ii) Cluster (there are 5 clusters with 7, 7, 9, 10, 16 nodes respectively where each cluster is bounded by a fixed square area), each with 49 nodes deployed in $15m \times 15m$ area. The communication radius for each node is $4m$. The random and cluster topologies are newly generated (new seed) for each trial run. The default number of members chosen as indirect pinger was 3, and protocol period was 20 time units. The suspicion timeout, `ACTIVE_TIMEOUT` and `PASSIVE_TIMEOUT` were set as 160, 80, and 400 time units in the experiments respectively. Each data point reflects data from 1000 independent runs. In every period, the probability to apply passive-feedback is 10%.

We define *first detection time* as the time gap between a failure occurring and the first non-faulty node detecting this failure (after suspicion timeout). We measure *dissemination time*, the time for all nodes to know about a failure after the first detection. Fig. 4.1 stacks dissemination time atop detection time. For a fair comparison, instead of raw first detection time, in (only)

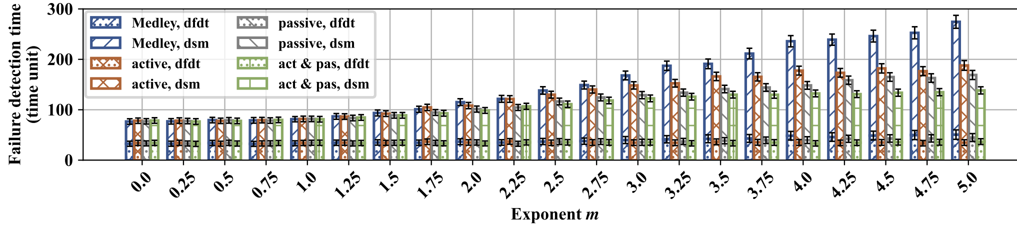


Figure 4.1: **Differential first detection time (dfdt), stacked dissemination time (dsm). Various m and optimizations. Random topology.**

this plot we use *differential first detection time (dfdt)* = (*first detection time*) minus (*minimum detection time*). The (theoretical) *minimum detection time* in our deployment is 180 time units: a sum of detection timeout of 20 time units and suspicion timeout of 160 time units.

We find that active-feedback (only) is the most effective at reducing *dfdt* by up to 31.1%. Active and passive together reduce by 27.4% and passive-feedback-only by 11.5%. Combining both active and passive provides 54.6% reduction in dissemination time (*dsm*), with passive-only at 44.5% and active-only at 31.6%. Intuitively, active-feedback-only offers the shortest and stablest first detection time even under high m, since it helps each *unlucky* node get frequent pings. For dissemination, intuitively, the combination of active-feedback spreading locally and passive-feedback spreading far away, is fastest. Overall, at m=3, we recommend combining active and passive, to reduce P95 *dfdt* by 31.2% and *dsm* by 47.2%.

Simultaneous Failures: We simultaneously fail 50% (randomly chosen) nodes (24 out of 49) in the Random topology. Fig. 4.2 shows the average first detection time. The lower and higher error bars are respectively the *earliest* and *latest* time any failure is detected, averaged across runs.

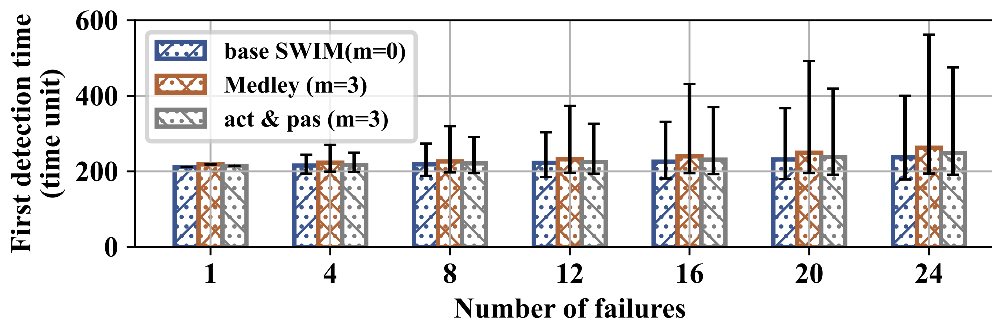


Figure 4.2: **Failure detection time under simultaneous failures (Random topology).**

The average (raw) first detection time of Medley and its variants rises gently as m and number of failures increase. As m rises, a failed unlucky node waits longer to become a ping target because pings stay local. Now, define the *detection gap* as the percentage by which detection time is prolonged under massive failure (50% nodes) vs. just a single failure scenario. In base SWIM (at $m = 0$), the detection gap is only 12.3%– due to the uniform randomness, a failed node has a high probability $1 - (\frac{48}{49})^{24} \simeq 39.0\%$ of being pinged each round after failure. In Medley, the detection gap is 20.6% due to localized pings and unlucky nodes' higher detection times. Applying active and passive feedback reduces the gap to 15.8%. Note that Medley's slightly longer detections come with massive bandwidth savings, which will be shown later in this section.

Domain Failure: Next, we explore the effect of massive failures in an area (e.g., connected to a power breaker). 49 nodes are located in five clusters in the square area of interest. Each run randomly fails a whole cluster.

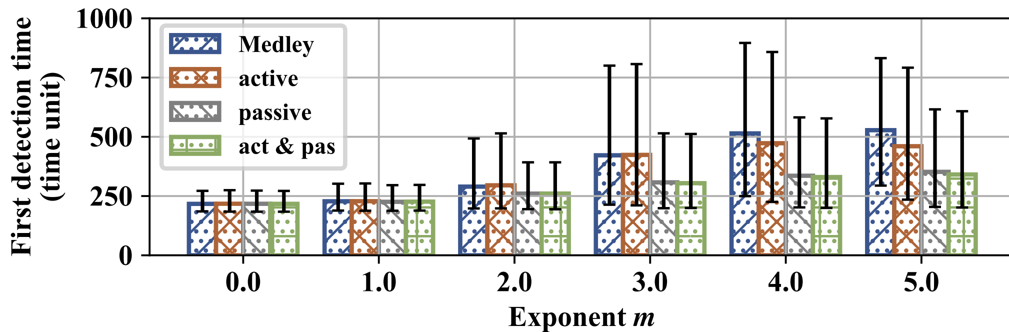


Figure 4.3: **Failure detection time under domain failures (Cluster topology).**

From Fig. 4.3 (bars similar to Fig. 4.2) we observe that the average first detection time stays low when $m < 2$, and as expected, it increases as m rises. The increase in detection time with m is because of the ping localization under higher m, implying that the typical way a detection proceeds at higher m is from the edges of the failed cluster towards the cluster's middle. In comparison, lower values of m would detect nodes near the middle of the failed cluster much quicker due to the higher probability of far-away non-faulty pingers. Under high m, both active and passive reduce detection time, with passive more effective since it provides a higher chance to detect nodes near the "middle" of the failed cluster.

Fig. 4.4 shows the CDF of the hop count of messages. Point(x, y) means $y\%$ of messages travel fewer than x hops. As expected, lower m (basic SWIM with uniform pinging) incurs far more hops, while Medley localizes traffic. Active feedback does not affect traffic much, since the ping probability modification occurs only among nodes with already-high pinging probabilities, i.e., already close to pinger. Passive feedback raises traffic as farther nodes are affected.

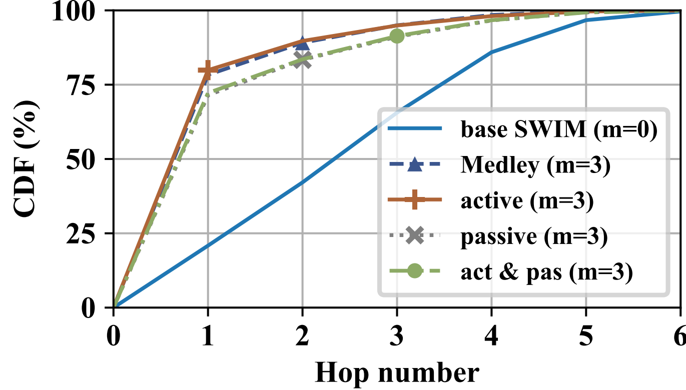


Figure 4.4: **CDF of hop numbers that messages travel under different strategies in Random topology.**

Since Medley's goal is to minimize both communication cost (messages sent, counting multiple hops) and detection time, we measure the square root of their product in Fig. 4.5, for Random topology. Each experimental run was identically long at 300K time units, so trends would remain unchanged if we replaced communication cost, i.e., messages, with bandwidth use, i.e., messages per second, or messages per second per hop. Medley's product cost (lower is better) falls quickly as m goes from 0 to 3, and then slowly rises. At lower m, Medley pings faraway nodes more frequently. Rising m increases detection time, yet the associated communication drop (due to localization) is faster. At higher m, communication cost reduction slows down, and thus detection time increase dominates. Compared to base SWIM ($m = 0$), Medley's product cost is 35.2% lower.

With active-feedback, Medley-F's product cost (under m=3) is 37.8% lower than base SWIM, as active-feedback lowers communication cost effectively. Applying passive-feedback, and active+passive, do not bring higher benefits, since passive sends faraway pings. However, product cost is still lowered by up to 30% and 31.2% respectively compared to basic SWIM. At $m > 3$, in all feedback-based strategies, communication reduction balances out detection

time increase.

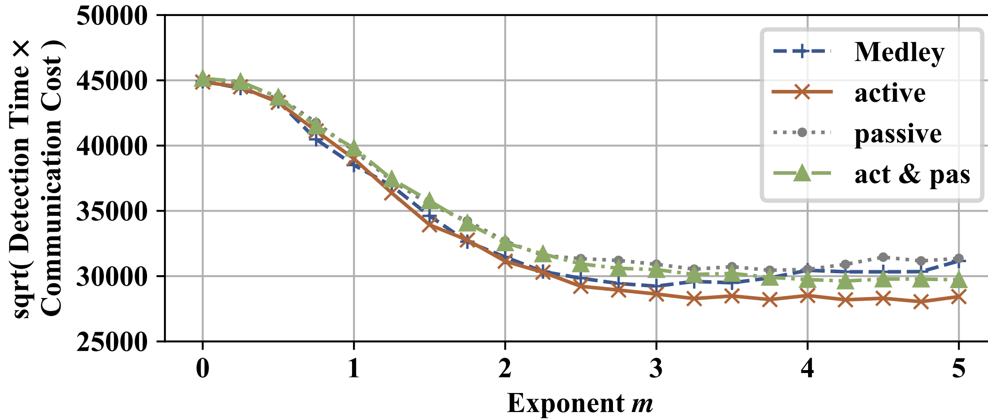


Figure 4.5: **Average failure detection time × Communication cost, Square Root (Random topology). Lower is better.**

4.1.2 Scalability

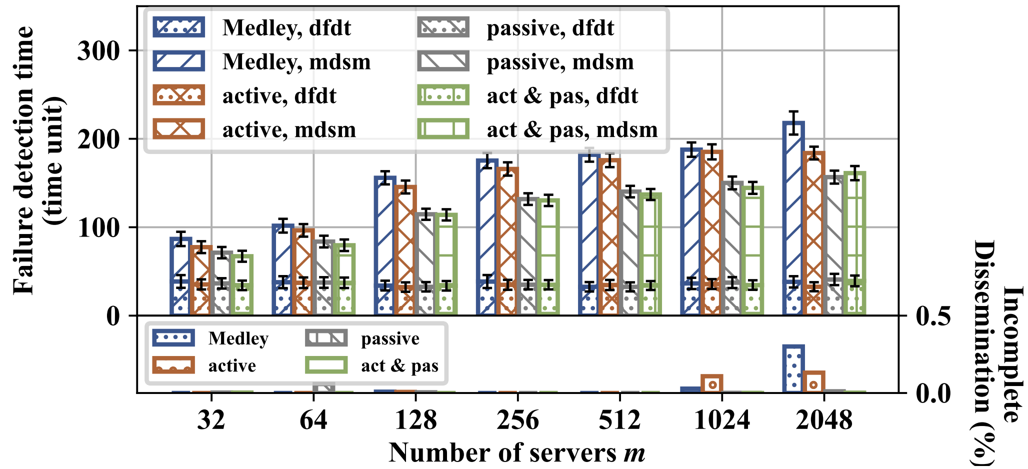


Figure 4.6: **Differential first detection time (dfdt), and stacked median dissemination time (mdsm), under different network sizes. For transparency, lower plot shows % nodes that does not receive dissemination before simulation ends.**

We evaluated the scalability of Medley and Medley-F up to $N = 2048$ nodes under $m = 3$, Random topologies, and in a square area, with fixed density of 0.22 node/sq meter. Fig. 4.6 shows: a) *dfdt*: first detection time (same as Fig. 4.1), and b) *msdm*: median dissemination time (stacked atop

dfdt). At large N, runs took long (e.g., N=2048 took 40 min per run, and thus 5 days for the full experiment), and so we truncate these experiments. While Medley is complete, for full transparency of results we also show the incomplete dissemination (due to experiment truncation) in the lower part of the plot. We plot the median and also plot standard deviation bars. Each data point in the plot is from 1000 runs, with the exception of 200 simulation runs at $N = 2048$.

We observe that: a) detection time is constant and insensitive to system size, b) median dissemination time increases logarithmically with system size (note the logarithmic x axis), and c) both active and passive strategies reduce dissemination time, with passive being both faster and able to reduce incomplete dissemination.

4.1.3 Performance Under Mobility

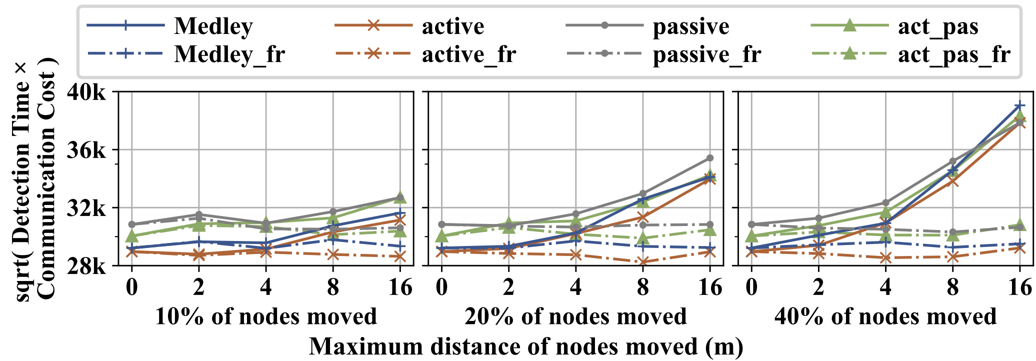


Figure 4.7: **Failure detection under Mobility.** *Solid lines show Medley operating with stale internal data structures after movement, while dotted lines (_fr) represent Medley with fresh data structures after movement.*

While Medley is intended for static topologies, we show that it is tolerant to moderate amounts of mobility. The experiment in Fig. 4.7 starts with 49 nodes in a random topology over a $15m \times 15m$ area. Each run starts Medley in a new topology and reaches a steady state. Then we instantly move a fraction of the nodes in random directions by distances randomly uniformly chosen among $[0, min(X$ meters, distance to edge of square)] (X is the x axis value on the plots). With this move, we *do not* update Medley's internal topology-related data at any nodes: distances, ping probabilities, etc., all remain stale from the pre-move topology. However, routing tables

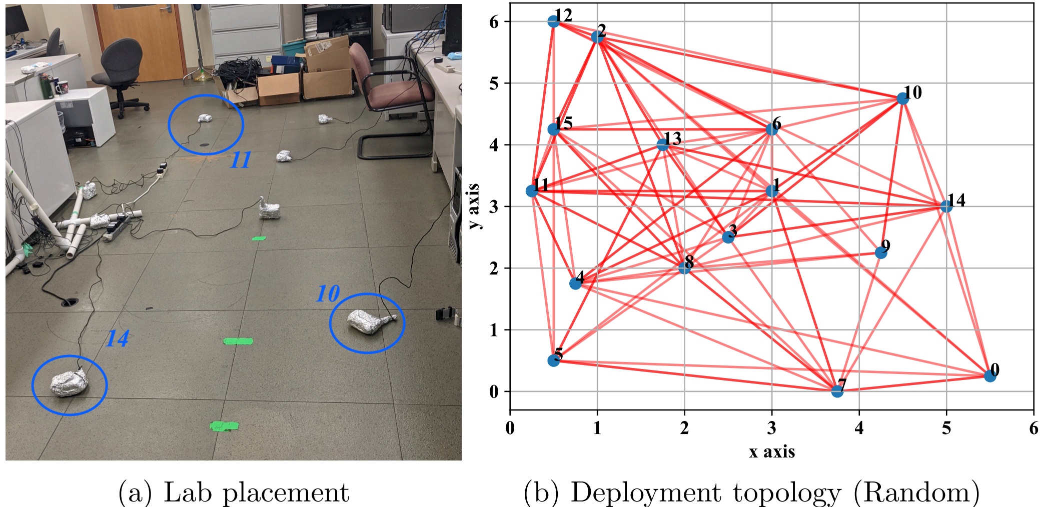
are updated post-move, as would be expected in a mobile network so that packets can be routed correctly.

In Fig. 4.7, the solid lines show the post-move performance of Medley, operating still with stale (pre-move) internal data structures. Dotted lines show Medley with corrected post-move internal data. Essentially, the gap between the solid and dotted lines shows the effect of Medley continuing with stale data structures. We observe that: a) at small mobility up to X=4 meters, even with up to 40% of nodes moved, stale data does not affect Medley performance; b) when few nodes move (10% plot), larger mobility distances can be tolerated—the metric rises by at most 7.1% at X=16; c) when more nodes move (40% plot), metric degradation is worse at 35% at X=16. Overall, we conclude that: 1) moderate mobility degrades Medley performance only moderately, and 2) Medley continues offering low communication and detection times even if its internal data (ping distances and hence ping probabilities) remain stale.

4.2 Deployment

We implemented a prototype of Medley in the Raspberry Pi (RP) 4 [63] environment. Our Java implementation was around 3000 lines of code, under Raspbian 4.19. We deployed Medley in a network of 16 IoT devices in our lab space. Figs. 4.8a and 4.8b show a photograph and a map of one of our topologies. This random topology was in a $6m \times 6m$ area (grid lines only for reference purposes). Each device was a Raspberry Pi 4 model B, with 2GB LPDDR4 RAM and Broadcom BCM2711, 1.5 GHz quad-core Cortex-A72 CPU. While Medley works modularly with any ad-hoc routing protocol, for concreteness we use OLSR routing [64] due to its ease of configurability for Pi4s, and popularity in discussion forum posts. Since the signal strength of Pis were too strong to make multi-hop routing with respect to the limited deployment area, we attenuated each Pi by both: a) consistently wrapping in aluminum foil, and b) setting transmit power to 15 dBm, to force more multi-hop transmissions. Red lines Fig. 4.8b is a screenshot of routine paths. Prior to these experiments, we performed benchmark experiments to verify that this attenuation was stable and consistent across Pis.

From Fig. 4.9 we observe that failure detection time and dissemination time



(a) Lab placement (b) Deployment topology (Random)

Figure 4.8: **Topology of Raspberry Pi deployment.**

both increase as m becomes larger. (The plot used 32 data points per failure, with average and standard deviation shown.) This is because disseminating failure information of unlucky nodes (e.g., nodes 0, 9 in Fig. 4.8) takes a while since spatial pinging (hence piggybacking of failure information) stays largely local especially at high m. Similar to simulation results, both active-feedback and passive-feedback produce benefits for first detection time and dissemination time. From the simulation (Sec. 4.1.1), we expected active-feedback to work the best for first detection time and passive-feedback to be effective on dissemination latency reduction. In the deployment, active-feedback and applying both strategies do act as expected. However, at high m, passive-feedback performs poorly on dissemination time, because benefits of multi-hop dissemination do not emerge in our smaller deployment scales. passive-feedback may only be more preferable at larger scales.

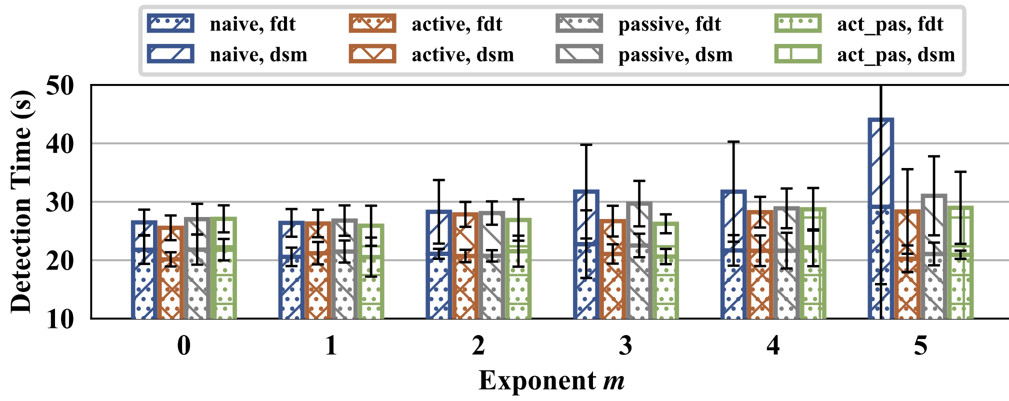


Figure 4.9: **First failure detection time and dissemination time for Raspberry Pi experiments.**

CHAPTER 5

CHURN-TOLERANT LEADER ELECTION

In this chapter, we present our churn-tolerant leader election protocols, which can be run on top of any weakly-consistent membership. Sec. 5.1 presents the motivation. Sec. 5.2 presents the detailed protocol designs, and Sec. 5.3 analyzes the protocols formally.

5.1 Motivation

Classical solutions to leader election [49, 23, 55, 32, 54, 56, 57, 65, 66] assume a "stable" membership (i.e., "strongly-consistent membership"), wherein every node knows all other nodes, and, if any node joins, leaves, or fails, the underlying membership protocol *sends an instantaneous update* to all non-faulty nodes. However, inconsistencies in membership lists are unavoidable in asynchronous edge systems since an update (node join, leave, or failure) cannot instantly propagate to all nodes: network latencies are non-zero and vary, while timeouts for failure detection (e.g., via heartbeating [25] or ping-ack protocols [28, 13]) are not synchronized across nodes.

We aim to solve leader election in such *churned* edge environments where nodes continuously join, leave, and fail (by crashing only [1]). Concretely, our leader election protocols can be layered atop an arbitrary *weakly-consistent* membership protocol such as gossip-style heartbeating [25], SWIM [28], or Medley [13]. These weakly-consistent membership protocols detect failures and spread membership updates quickly, yet they only provide *eventual* consistency guarantees, i.e., a membership update (join, leave, or failure) is received only eventually at all non-faulty nodes, and in an arbitrary order. Due to their scalability, weakly-consistent membership protocols are an appropriate choice for edge computing settings.

[1] Fail-stop model only. This paper does not consider Byzantine failures.

The problem of churn-tolerant leader election is non-trivial. Any leader election protocols should satisfy both the liveness and safety property:

Definition 3. ***Liveness:*** *Protocol terminates, i.e., each alive node eventually elects a leader (sets its local leader variable to a non-null value).*

Definition 4. ***Safety:*** *At the end of the leader election, each alive node only sets its local leader variable to that unique non-faulty node which has the lowest hash of all non-faulty nodes currently in the system.*

Given strongly-consistent membership, a group can quickly elect a leader: each node hashes all IDs present in its local membership list and selects as leader node whose ID has the lowest hash [67, 49, 13]. With identical and correct membership lists (strong), everyone elects the same leader, without messages. However, if membership lists are inconsistent (weak), the would-be leader may be unknown at some nodes, thus multiple leaders may be elected, violating safety. Thus, a new churn-tolerant variant of this protocol is needed. (For simplicity the rest of this paper uses the lowest hash ID as the leader selection criteria, but this can be replaced by an arbitrary attribute of choice, and our results still apply.)

5.2 Protocol Design

In this section, we present our four leader election protocols for churned settings [18]. We first define the churn value c that is used by our protocols. Denote the total number of nodes in the system as N. Let a given node M_i be missing in the membership lists of c_i other nodes. We assume that the maximum of all c_i values is bounded from above by a known value c (analogous to the classical assumption of f failures), i.e.,

$$max(\{c_i | 1 \leq i \leq N\}) \leq c$$

We aim to design an election protocol that provides safety for a given maximum value of c—we call such an election as a *c-tolerant election* protocol (or more precisely *(c,f)-tolerant*). Our notion of *c-tolerance* handles missing entries in membership lists, thus generalizing the traditional *failure (f)-tolerant*

protocol that deals with extra entries (nodes that are failed but not yet detected).

In practice, values of c are in fact small. Table 5.1 shows 5-minute runs of the Medley failure detector [13] with 3 different topologies, system sizes, and message drop rates. We observe that the calculated $c(= max(\{c_i | 1 \leq i \leq N\}))$ values are small. For message loss rates at or under 5%, c values never exceed 10% of N. Hence we assume the membership graph is strongly connected, not partitioned, and that c is small.

Table 5.1: *c values (from 5-minute Medley [13] runs) stay small across different configurations*

Topology Type	Topology Size	Message Drop Rate	c
Grid	49	0.05	4
Cluster	49	0.05	4
Random	49	0.05	4
Random	49	0.1	7
Random	49	0.15	10
Random	49	0.2	13
Random	32	0.05	2
Random	64	0.05	5
Random	128	0.05	9
Random	256	0.05	17

5.2.1 Base Protocol

We first observe that in order to minimize message complexity, the initiator must communicate with the least number of nodes, to safely make a decision about who the leader is.

Assume node M_i is the *would-be (or presumptive) leader* (i.e., the non-faulty node with the lowest hash in the system, though other nodes may not know it yet). Assume no failed nodes at first. By our assumption, M_i may be missing in up to c other nodes' membership lists. Thus, our protocol has the initiator send a QUERY message to at least $(c+1)$ nodes, selected arbitrarily, which then send back a RESPONSE message with their lowest known hash node (from their respective membership lists). By definition, at least one of the $(c+1)$ responses will contain the would-be leader M_i.

Next, with up to f failures, some of the nodes contacted by the initiator may not respond. Thus we increase the number of nodes that the initiator contacts to $(c+f+1)$ nodes (selected arbitrarily) so that the initiator receives at least $(c+1)$ responses.

Once the initiator calculates the leader, it sends the would-be leader a NotifyLeader message. Upon receiving this, the leader knows it is the leader, and it multicasts a Leader message to the entire group.

This *Base Protocol* is depicted formally in Algorithm 2. The Timeout in line 14 is set based on the expected round-trip time so that if the would-be leader failed during the election run, a new election is started.

Algorithm 2 Base Protocol

```
1:  function InitiateElection // At Initiator
2:      id ← ∞, receivedIds ← ∅
3:      Send Query to arbitrary c + f + 1 nodes
4:      while size(receivedIds) < c + 1 do
5:          if no new Response after Timeout then
6:              Send Query to c + f + 1 − size(receivedIds) nodes excluding
    receivedIds
7:          end if
8:          newId ← Response from node i
9:          receivedIds.add(i)
10:         id ← min(id, newId)
11:     end while
12:     Send NotifyLeader to node id
13:     if No Leader message from id after Timeout then
14:         InitiateElection()
15:     end if
16: end function
17:
18: function RecvMessage(msg from node i) // Any Node
19:     if msg is of type Query then
20:         Send Response (← lowest hash node) to node i
21:     else if msg is of type NotifyLeader then
22:         Multicast Leader
23:     else if msg is of type Leader then
24:         Mark node i as leader
25:     end if
26: end function
```

Fig. 5.1 depicts an example of the Base Protocol with 4 nodes. We use $c = 2$, $f = 0$, and node 3 is the initiator.

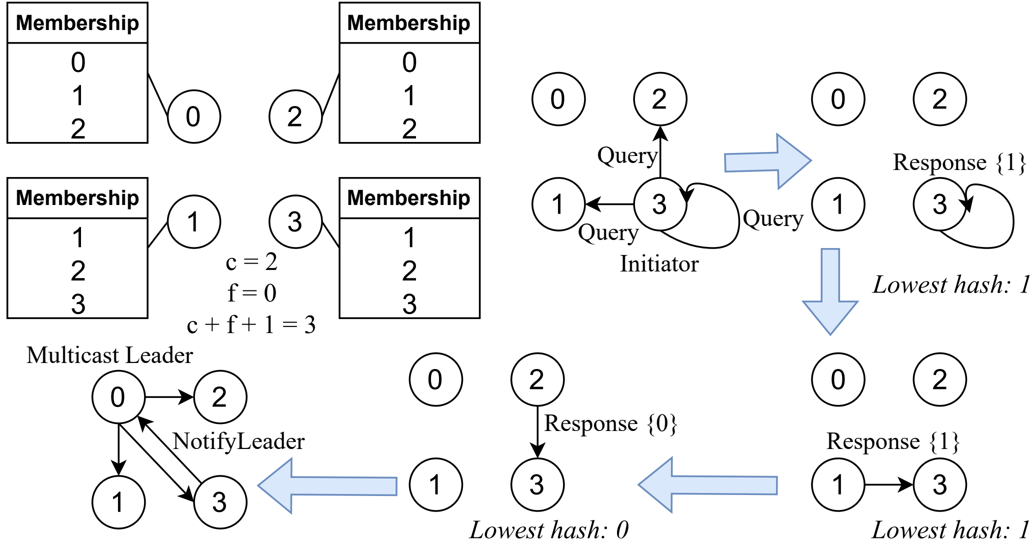


Figure 5.1: *Base Protocol Example*

5.2.2 Optimistic Protocol

The Base Protocol can be slow in the common case when membership lists are *nearly* consistent. To converge quickly in this optimistic case, we present the Optimistic variant shown in Algorithm 3. The key idea is for the initiator to "stream" the leader calculation—whenever a new Response is received, the leader is re-calculated and, if it changes, a new NotifyLeader message is sent to the newly elected leader (which then multicasts Leader to the group). Nodes may receive multiple Leader messages and use only the lowest hash value node as the leader.

If membership lists are nearly consistent, the Optimistic Protocol converges quickly, because one of the early Response messages received at the initiator will contain the would-be leader and further NotifyLeader messages will not be sent.

5.2.3 Preferred Protocol

Some applications additionally need to elect the lowest hash leader who is also *healthy*, i.e., is being suspected less often by other nodes' failure detectors as a false positive. Our third protocol, called Preferred Protocol (Algorithm 4), has each Response message contain both the top (parameterized) y unhealthy nodes and the lowest (parameterized) x hash nodes. (In our

Algorithm 3 Optimistic Protocol

```
1: function InitiateElection // At Initiator
2:     id ← ∞, receivedIds ← ∅
3:     Send Query to arbitrary c + f + 1 nodes
4:     while size(receivedIds) < c + 1 do
5:         if no new Response after Timeout then
6:             Send Query to c + f + 1 − size(receivedIds) nodes excluding
   receivedIds
7:         end if
8:         newId ← Response from node i
9:         receivedIds.add(i)
10:        if newId < id then
11:            id ← newId
12:            Send NotifyLeader to node id
13:        end if
14:    end while
15:    if No Leader message from id after Timeout then
16:        InitiateElection()
17:    end if
18: end function
19:
20: function RecvMessage is identical to the Base Protocol
```

implementation, typical values are $x = y = 5$.) The health of a node can be calculated in an application-defined way: possibilities include metrics such as false positive count [25] or suspicion counts (in SWIM [28] or Medley [13])—these metrics are readily available from the membership protocol itself. The initiator calculates the set *leaders*: by taking the union of all received lowest hash ID nodes, and removing the union of all received unhealthy nodes. The initiator uses the lowest hash node from *leaders* as the leader (the rest of the protocol remains unchanged). If $leaders = \emptyset$, the protocol retries by increasing x (GetNextX) and reducing y (GetNextY).

5.2.4 Hybrid Protocol: Combining Optimistic and Preferred

Optimism and Preference are orthogonal and our Hybrid Protocol in Algorithm 5 combines both. Whenever the initiator receives a new Response, it calculates the lowest hash node with unhealthy nodes excluded. If this results in a leader change, a new NotifyLeader is sent to the newly elected leader. This protocol converges quickly if membership lists happen to be mostly consistent system-wide.

Example—Hybrid Protocol: In Fig. 5.2 there are 5 nodes (with respective hash values $0, 1, 2, 3, 4$), and the parameters are set as: $x \leftarrow 2, y \leftarrow 2, c \leftarrow 2, f \leftarrow 0,$ and $c + f + 1 \leftarrow 3$. The unhealthiness metric takes integer values. Consider the following snapshot of the system operation. For each line, the first number denotes which node's membership list we are looking at, and the corresponding dictionary of (key : value) pairs indicate (node : unhealthiness metric) pairs:

$$0 : \{1 : 2, 2 : 1, 3 : 4, 4 : 0\}$$

$$1 : \{0 : 5, 2 : 0, 3 : 2, 4 : 1\}$$

$$2 : \{0 : 2, 1 : 2, 3 : 3, 4 : 1\}$$

$$3 : \{0 : 3, 1 : 0, 2 : 2, 4 : 1\}$$

$$4 : \{0 : 0, 1 : 1, 2 : 0, 3 : 2\}$$

Suppose node 4 is the initiator and it sends Query to nodes $0, 1, 4$. They reply back with their top $y = 2$ unhealthy nodes and the lowest $x = 2$ hash

Algorithm 4 Preferred Protocol

```
1: function InitiateElection(x, y) // At Initiator
2:     candidates, excludes, receivedIds ← ∅
3:     Send Query{x, y} to arbitrary c + f + 1 nodes
4:     while size(receivedIds) < c + 1 do
5:         if no new Response after Timeout then
6:             Send Query to c + f + 1 − size(receivedIds) nodes excluding
   receivedIds
7:         end if
8:         {candidate, exclude} ← Response from node i
9:         receivedIds.add(i)
10:        candidates ← candidates ∪ candidate
11:        excludes ← excludes ∪ exclude
12:    end while
13:    leaders ← candidates − excludes
14:    if leaders = ∅ then
15:        x ←GetNextX(x), y ←GetNextY(y)
16:        InitiateElection(x,y)
17:        return
18:    end if
19:    id ← min(leaders)
20:    Send NotifyLeader to node id
21:    if No Leader message from id after Timeout then
22:        InitiateElection(x,y)
23:    end if
24: end function
25:
26: function RecvMessage(msg from node i) // Any Node
27:    if msg is of type Query then
28:        {x, y} ← Query
29:        exclude ← top y unhealthy nodes
30:        candidate ← x lowest hash (excludes exclude)
31:        Send Response{candidate, exclude} to node i
32:    else if msg is of type NotifyLeader then
33:        Multicast Leader
34:    else if msg is of type Leader then
35:        Mark node i as leader
36:    end if
37: end function
38:
39: function GetNextX(x): return min(N, x + 1)
40: function GetNextY(y): return max(0, y − 1)
```

Algorithm 5 Optimistic and Preferred (Hybrid) Protocol

1: **function** InitiateElection(x, y) // At Initiator
2: $candidates, excludes, receivedIds \leftarrow \emptyset, id \leftarrow \infty$
3: Send Query$\{x, y\}$ to arbitrary $c + f + 1$ nodes
4: **while** $size(receivedIds) < c + 1$ **do**
5: **if** no new Response after Timeout **then**
6: Send Query to $c + f + 1 - size(receivedIds)$ nodes excluding $receivedIds$
7: **end if**
8: $\{candidate, exclude\} \leftarrow$ Response from node i
9: $receivedIds.add(i)$
10: $candidates \leftarrow candidates \cup candidate$
11: $excludes \leftarrow excludes \cup exclude$
12: $leaders \leftarrow candidates - excludes$
13: **if** $leaders = \emptyset$ **then**
14: $x \leftarrow$GetNextX(x), $y \leftarrow$GetNextY(y)
15: InitiateElection(x,y)
16: **return**
17: **end if**
18: $leaderId \leftarrow min(leaders)$
19: **if** $id \mathrel{!}= leaderId$ **then**
20: $id \leftarrow leaderId$
21: Send NotifyLeader to node id
22: **end if**
23: **end while**
24: **if** No Leader message from id after Timeout **then**
25: InitiateElection(x,y)
26: **end if**
27: **end function**
28:
29: **function** RecvMessage, GetNextX, and GetNextY are identical to the Preferred Protocol

nodes. Below is one possible execution outcome of the Hybrid Protocol:

- Initiator receives RESPONSE from node 4

  - $excludes \leftarrow \{3:2, 1:1\}$

  - $candidates \leftarrow \{0, 2\}$

  - $leaders \leftarrow \{0, 2\}$

  - Node 0 becomes the tentative leader and the initiator sends NOTIFYLEADER to node 0

  - Node 0 multicasts LEADER to everyone

- Initiator receives RESPONSE from node 0

  - $excludes \leftarrow \{3:6, 1:3\}$

  - $candidates \leftarrow \{0, 2\}$

  - $leaders \leftarrow \{0, 2\}$

  - Node 0 remains the tentative leader

- Initiator receives RESPONSE from node 1

  - $excludes \leftarrow \{3:8, 0:5, 1:3\}$

  - $candidates \leftarrow \{2\}$

  - $leaders \leftarrow \{2\}$

  - Node 2 becomes the tentative leader and the initiator sends NOTIFYLEADER to node 2

  - Node 2 multicasts LEADER to everyone

- At this point the initiator has received all the RESPONSE messages, thus the confirmed leader is node 2

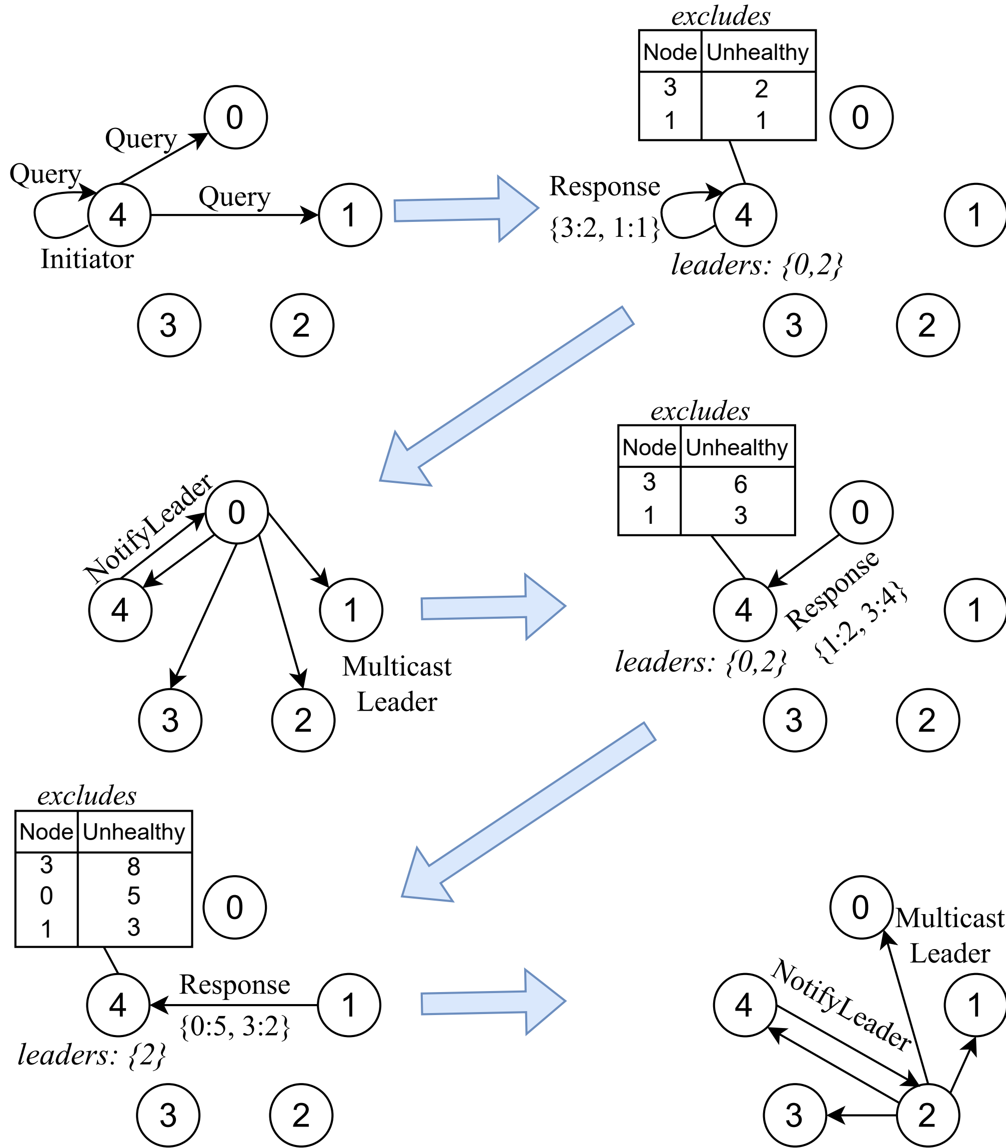


Figure 5.2: *Hybrid Protocol Example*

5.3 Formal Analysis

We formally analyze the properties of the four protocols. Readers may skip this section without loss of continuity—for such readers, we provide here a summary of our findings:

1. The Base Protocol and Optimistic Protocol both satisfy Safety and elect the lowest hash ID node as the leader. (Note that the Preferred Protocol and Hybrid Protocol may not elect the lowest hash ID leader if it is unhealthy.)

2. All our four protocols (Base, Optimistic, Preferred, Hybrid) satisfy Liveness: they complete and elect a leader.
3. The Base Protocol is optimal in message complexity.
4. The Preferred Protocol and Hybrid Protocol do not elect top unhealthy nodes as leaders, with high probability.

Theorem 1. *(Safety) Both the Base Protocol and Optimistic Protocol satisfy Safety (as defined in Sec. 2.3)).*

Proof. We prove this by contradiction. Assume the would-be leader is the non-faulty node M_i (with lowest hash) but instead there exists at least one node that sets another node M_j ($j \neq i$) as its confirmed leader. This can only occur if all the initiator's received $(c+1)$ responses contain M_j but not M_i. However, M_i is missing in c_i nodes' membership lists, and hence this implies that $c_i > c$, which is a contradiction to the definition of c. □

Theorem 2. *(Liveness) If the initiator and would-be leader remain alive, all our four protocols (Base, Optimistic, Preferred, Hybrid) satisfy Liveness (as defined in Sec. 2.3)).*

Proof. We prove the theorem by contradiction. Assume the leader is never elected and the protocols do not finish. This may occur due to three reasons:

(1) If the initiator does not receive at least $(c+1)$ Response messages: Since the initiator sends Query to $(c+f+1)$ arbitrary nodes and the maximum number of failed nodes is f, at least $(c+1)$ nodes are alive and will send back a Response. Because messages are eventually delivered, the initiator receives at least $(c+1)$ responses.

(2) If the initiator never receives Leader message from the lowest hash node: Since multicast is reliable, this happens only if:

- The would-be leader is dead: in this case, the election will be restarted after the initiator times out; or
- The would-be leader does not receive NotifyLeader: this cannot happen as messages are delivered eventually.

(3) For the Preferred Protocol and Hybrid Protocol, if *leaders* set (Algorithm 4 and 5) is empty: Then the protocol will restart with the new x and y values. If this keeps reoccurring then according to GetNextX

and GetNextY, eventually we have $x = N$ and $y = 0$. Then, *candidates* contains all the N nodes and *excludes* is empty, which makes $leaders = candidates - excludes = candidates$ non-empty. Thus, there will always be a leader elected in this case.

Hence all four protocols satisfy Liveness. □

Lemma 1. *Assume unicast is reliable and nodes don't fail. Then, the Base Protocol involves ($2 \cdot (c + f + 1) + 1$) unicasts and 1 multicast.*

Proof. The leader multicasts Leader only once at the end of the election process. The initiator sends $c + f + 1$ Query messages to arbitrary nodes, and these nodes will reply back with $c + f + 1$ Response messages. The initiator also sends another NotifyLeader to the would-be leader. □

Theorem 3. *(Message Optimality) Assume unicast is reliable and nodes don't fail. Then (among all initiator-based election protocols) the Base Protocol is optimal in message complexity in order to satisfy Safety.*

Proof. For any leader election protocol, we need at least 1 multicast message to let everyone know the new leader. The single NotifyLeader unicast is also necessary since the leader needs to be informed by the initiator that it is the leader. Thus, at least 1 multicast and 1 unicast are required by any election protocol.

Next, suppose there exists a leader election protocol that works with less than $2 \cdot (c + f + 1)$ unicast messages. We prove by contradiction that this would violate Safety. Suppose only $(c+f)$ nodes are contacted by the initiator (and thus $2 \cdot (c + f)$ total unicasts). If f of these nodes fail, the initiator will only receive c replies. In the worst case scenario, the would-be leader M_i (non-faulty node with the lowest hash) is absent in c_i node's membership lists, and these are the $c = c_i$ nodes that send Response messages to the initiator. This will result in a different node than M_i being elected as leader, thus violating Safety. Therefore, we conclude that at least $2 \cdot (c + f + 1)$ unicasts are necessary.

Together with Lemma 1, this proves the theorem. □

Since the Preferred Protocol and Hybrid Protocol may not elect the lowest hash ID leader if it is unhealthy, we define:

Definition 5. ***Preference:*** *The elected leader does not belong to the system-wide top y unhealthy nodes.*

Theorem 4. *(Preference w.h.p.) With only a logarithmic number of messages, the Preferred Protocol and Hybrid Protocol satisfy Preference with high probability (w.h.p.).*

Proof. Denote r as the number of RESPONSE messages the initiator receives. Denote p as the average percentage of nodes that belongs to system-wide top-y unhealthy nodes but do not exist in an arbitrary RESPONSE message (top-y unhealthy nodes in local membership lists). Then, the probability that a given top-y system-wide unhealthy node will be in at least one of the r RESPONSE is $1-p^r$. To achieve this with high probability $> (1-\frac{1}{N})$, we need $r > \log_{1/p}(N)$. This implies the Preferred Protocol and Hybrid Protocol only require a logarithmic number of RESPONSE messages to satisfy Preference w.h.p. □

Theorem 4 in Practice: Practically, we observed that the value of p stays small. Via multiple runs of Medley [13], we find that p only depends on the topology size N (and is independent of topology and message drop rate). In fact, Table 5.2 shows that p converges to 0.5 as N increases. Further, the r values never exceed $(c+1)$. Since the initiator receives at least $(c+1)$ RESPONSE messages, this suffices to satisfy *Preference* w.h.p.

Table 5.2: *r and $(c+1)$ values with different topology sizes*

Topology Size N	32	49	64	128	256
p	0.34	0.43	0.48	0.5	0.5
r	3	5	6	7	8
$c+1$	3	5	6	10	18

CHAPTER 6

LEADER ELECTION EXPERIMENTS

In this chapter, we present the experiment results of Churn-tolerant leader election protocols. The results include the comparison between the Base Protocol, Optimistic Protocol, Preferred Protocol, and Hybrid Protocol. Sec. 6.1 shows the simulation results, and Sec. 6.2 shows the Raspberry Pi deployment results.

6.1 Trace-Driven Simulation

The research questions addressed by our trace-driven simulation in this section are:

1. What is the bandwidth and leader election completion time for our four c-tolerant election protocols?
2. How well do the Base Protocol and Optimistic Protocol satisfy Safety?
3. Do the Optimistic Protocol and Hybrid Protocol shorten the completion/convergence time?
4. How healthy are the leaders elected by the Preferred Protocol and Hybrid Protocol?

We wrote and tested a custom simulator incorporating our four election algorithms. Our custom simulator allows us more agility to vary parameters than stock simulators (like NS-3) and also allows us to scale simulations better. In our simulator, messages can be dropped or delayed. Because we are dealing with ad-hoc routing topologies, packets are routed via Dijkstra's shortest path protocol.

Simulations are driven by real membership traces that we collected from running a weakly-consistent membership protocol, Medley (code obtained

and run from the authors of [13]). We configured the membership protocol to run with the identical network configuration (e.g., message drop rate, topology, etc.) as our leader election protocols. We collect membership traces only after its warm-up phase has finished. These traces are injected into our election protocols—our simulator continually reads the latest membership list at each node (along with information such as suspicion counts as a health metric) and updates it, and our election protocol implementation has to automatically cope with any changes.

Our default system size is 49 (we vary this). We evaluate using three topologies: Grid topology (default 7x7 grid), Random, and Cluster (total of 5 clusters with 7, 7, 9, 10, and 16 nodes respectively). Topologies with 49 nodes are simulated in a 15m x 15m space. Topologies with other sizes have a fixed node density of around 0.22 node/sq meter.

Each plotted data point is from 100 simulation runs using different seeds and different initiators. Message delay on each hop is randomly chosen in $(0, 50]$ (time units), and the communication range is 4 meters. Default parameter values in Sec. 5.2's pseudocode are: TIMEOUT = 500 time units, $x = 5$, and $y = 5$. We set the value of c according to Table 5.1.

In order to show the benefit of the Preferred Protocol and Hybrid Protocol, our network simulator has higher drop rates for messages sent and received by nodes with lower hash values. The consequence is that nodes with higher hashes will be healthier than nodes with lower hash. This forces the protocols to avoid electing the lowest hash node as the leader.

We show and discuss data for bandwidth, completion/convergence time, and leader health.

6.1.1 Bandwidth

Fig. 6.1, 6.2, and 6.3, show the bandwidth for our four protocols, measured as end-to-end bytes (summed across all hops for each packet's route).

Among our four protocols, we observe that while the bandwidth of the Base Protocol, Optimistic Protocol, and Preferred Protocol are similar, the Hybrid Protocol incurs relatively more bandwidth. This is because the unhealthiness information varies across nodes, and thus "top unhealthy" lists may look different at different nodes. Consequently (in the Hybrid Proto-

col), received RESPONSE messages at the initiator may frequently update the leader, leading to more LEADER multicast messages, and thus higher network bandwidth.

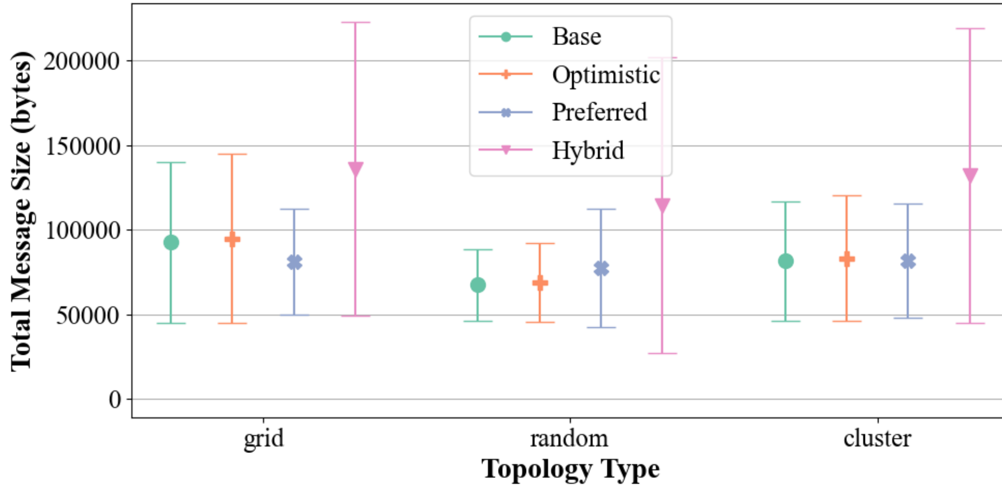


Figure 6.1: *Bandwidth with different topology types*

Bandwidth rises linearly with message drop rate and system size, as expected (Fig. 6.2 and 6.3). Larger system sizes or higher message drop rates naturally require higher values for c, thus increasing bandwidth via the $(c+f+1)$ QUERY and RESPONSE messages. Across different topology types, the comparative trends do not vary much (Fig. 6.1).

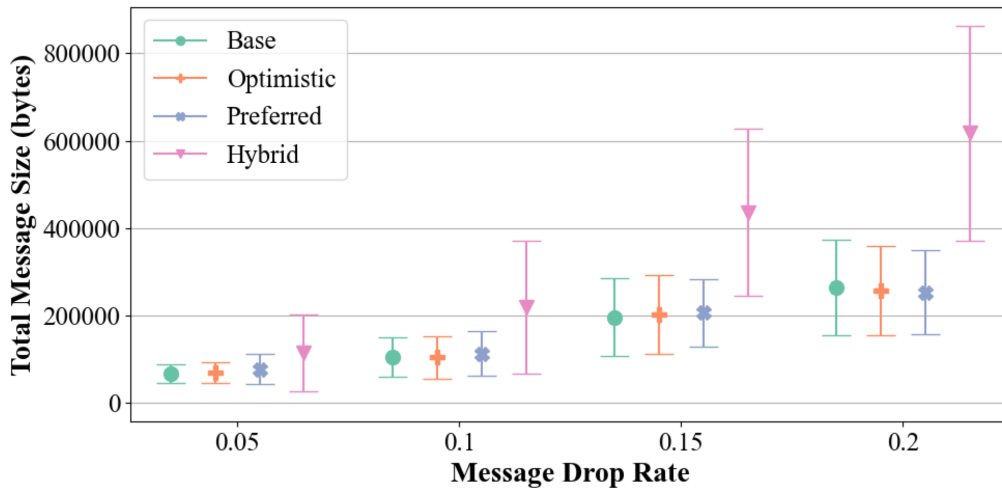


Figure 6.2: *Bandwidth with different drop rates*

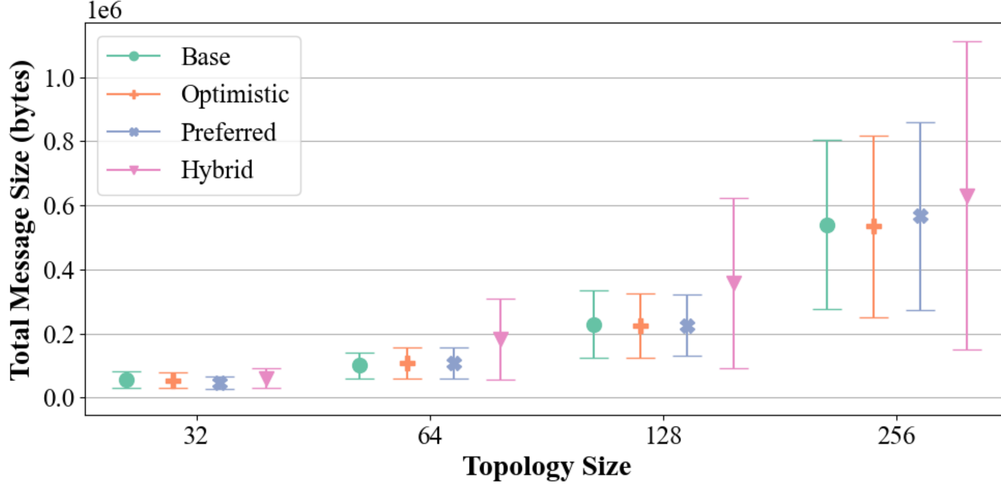


Figure 6.3: *Bandwidth with different topology sizes*

6.1.2 Completion Time (or Convergence Time)

The completion time of leader election is defined as the difference between when the initiator initiates the election and when the last non-faulty node knows the correct leader (i.e., receives LEADER). Fig. 6.4, 6.5, and 6.6 show the completion time across topology types, message drop rates, and system sizes.

We observe that the completion time of the Base Protocol and Preferred Protocol are large since both of them wait for all $(c+1)$ RESPONSE messages before notifying the leader. On the other hand, the completion time of the Optimistic Protocol is 42.6% less than the Base Protocol on average, and the completion time of the Hybrid Protocol is 33.3% less than the Preferred Protocol on average. These small completion times of the Optimistic Protocol and Hybrid Protocol indicate they are able to naturally leverage the existing consistency across membership lists and converge faster.

Fig. 6.5 and 6.6 show that, as expected, higher message drop rates and system sizes prolong completion times. Higher drop rates mean some messages (e.g., QUERY, RESPONSE) may be resent. Large system sizes mean higher values of c and thus longer wait time for the initiator to get $(c+1)$ responses.

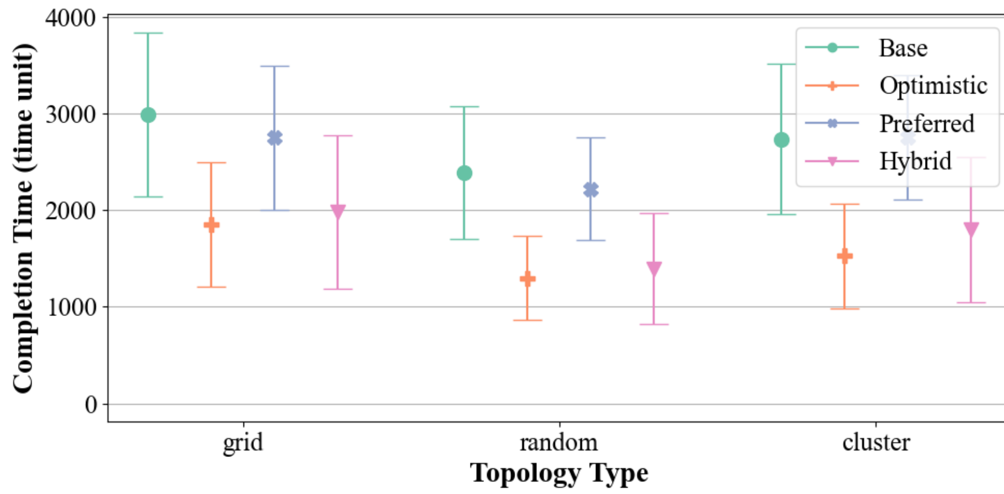


Figure 6.4: *Completion time with different topology types*

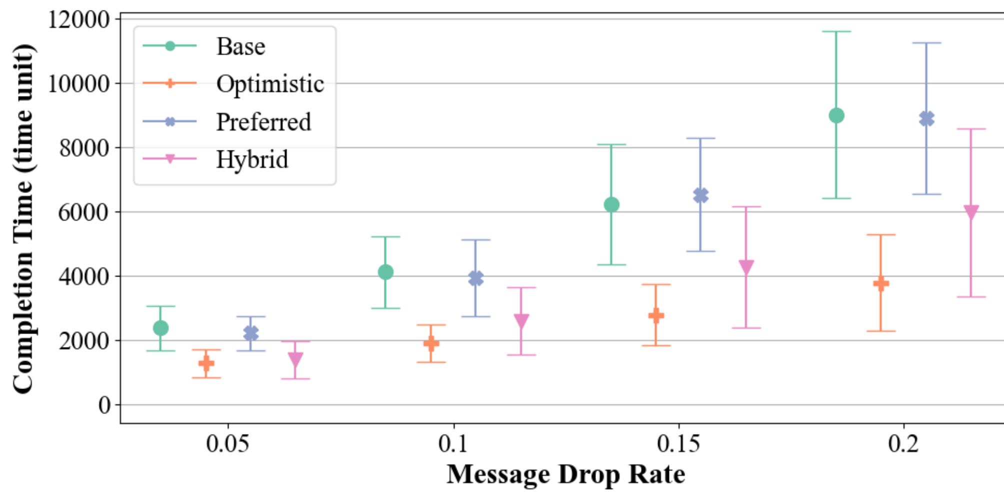


Figure 6.5: *Completion time with different drop rates*

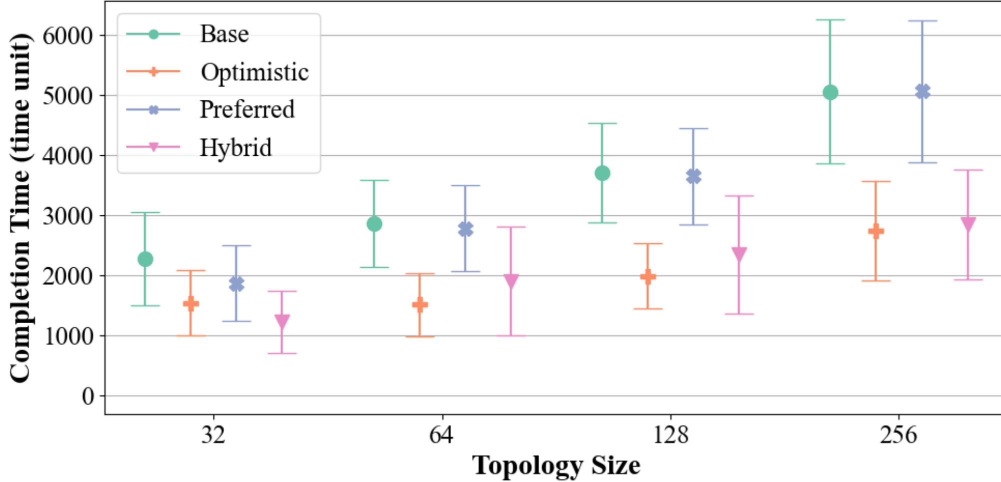


Figure 6.6: *Completion time with different topology sizes*

6.1.3 Leader Health

Recall that the Preferred Protocol and Hybrid Protocol (Sec. 5.2) are aimed at optimizing the health of the leader, i.e., attempt to elect a leader that is not among the top unhealthy nodes in the system, while the Optimistic Protocol and Hybrid Protocol are aimed at finishing quickly.

We measure the implications of these optimizations by measuring three different metrics: the number of times the leader is changed within the same election run, the unhealthiness of the elected leader, and the hash rank of the elected leader.

Leader Change Count: This is the *number of times that a different leader multicasts* LEADER *message within the same election run.* This metric measures the disruption to the application (running at nodes) which may need to take actions (e.g., contacting the leader) right after it recognizes a new leader. Fig. 6.7 shows that, as expected, the leader change count for both the Base Protocol and Preferred Protocol are small (= 1, since they only send one leader notification per run). The Optimistic Protocol has a small leader change count because membership lists are largely consistent. The Hybrid Protocol has the highest leader change count, increasing with message drop rate—28% worse than the Base Protocol at 0.05 message drop rate, and 157% worse at 0.2 drop rate. Hence, the tradeoff achieved by the Hybrid Protocol is frequent leader changes (during the election) vs. a higher quality leader (next paragraph).

Unhealthy Rank: This is the *system-wide unhealthiness rank of the elected leader*, where a higher value means the leader is healthier (0 represents the most unhealthy node and N represents the healthiest node in the system). Fig. 6.8 shows that: (a) the Base Protocol and Optimistic Protocol may be lucky and elect healthier leaders at low message drop rates (0.05) since many nodes are healthy, but as the drop rate rises to 0.1 and beyond, these protocols start electing largely unhealthy leaders, and (b) the Preferred Protocol and Hybrid Protocol elect healthier leaders with 1.5× better healthiness ranks than the Base Protocol at 0.05 message drop rate and 28.5× at 0.2 message drop rate (the Optimistic Protocol is similar).

Hash Rank: This metric shows how close our protocol comes to electing the "best attribute" (lowest hash) leader. Concretely, it is the *rank of the leader node hash*, with 0 representing the lowest hash node (and N the highest hash). Fig. 6.9 shows that both the Base Protocol and Optimistic Protocol only elect node 0 (lowest hash ID), as expected. While the Preferred Protocol and Hybrid Protocol elect higher hash ranks, the hash rank values stay low and range from around 0.5 to 2.0 (out of large N)—this indicates that the Preferred Protocol and Hybrid Protocol are able to elect healthy leaders (as we saw in Fig. 6.8) while still being able to minimize the "best attribute" (hash ID) of the elected leader.

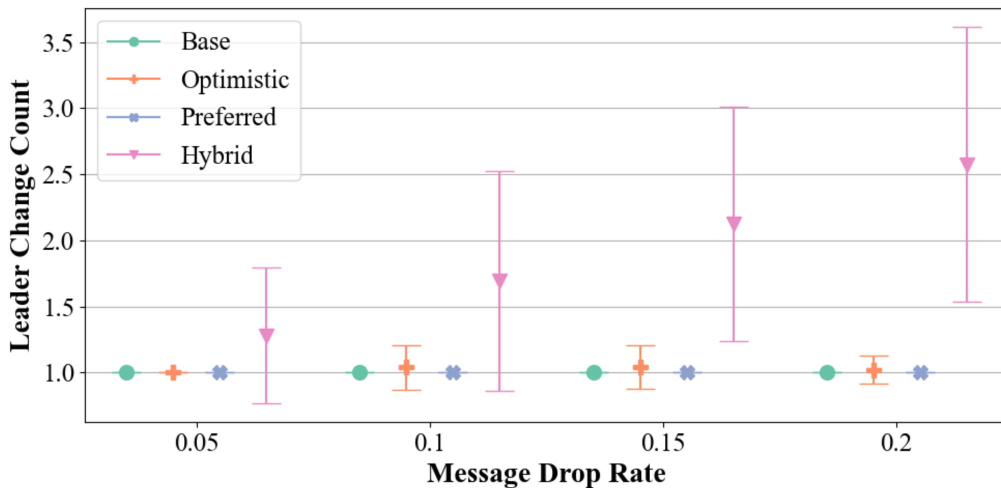


Figure 6.7: *Leader change count with different drop rates*

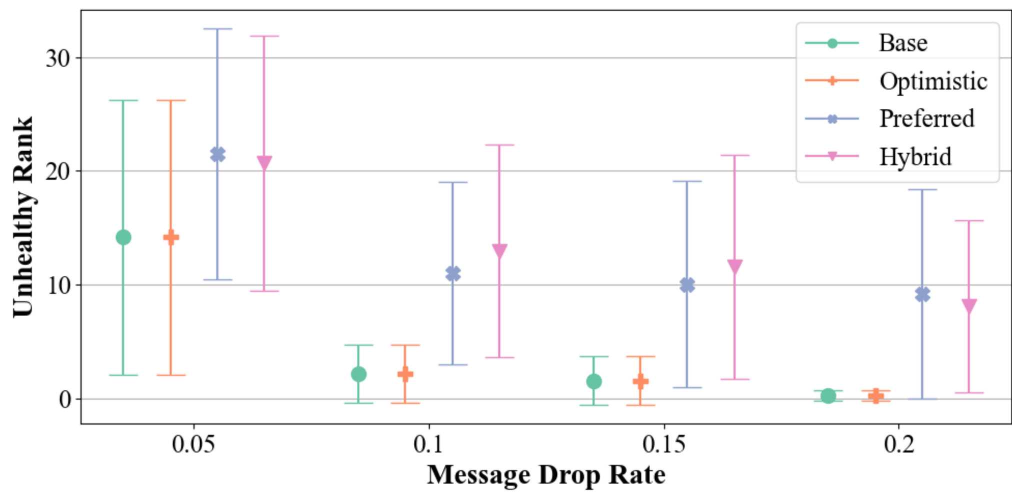


Figure 6.8: *Leader unhealthy rank with different drop rates*

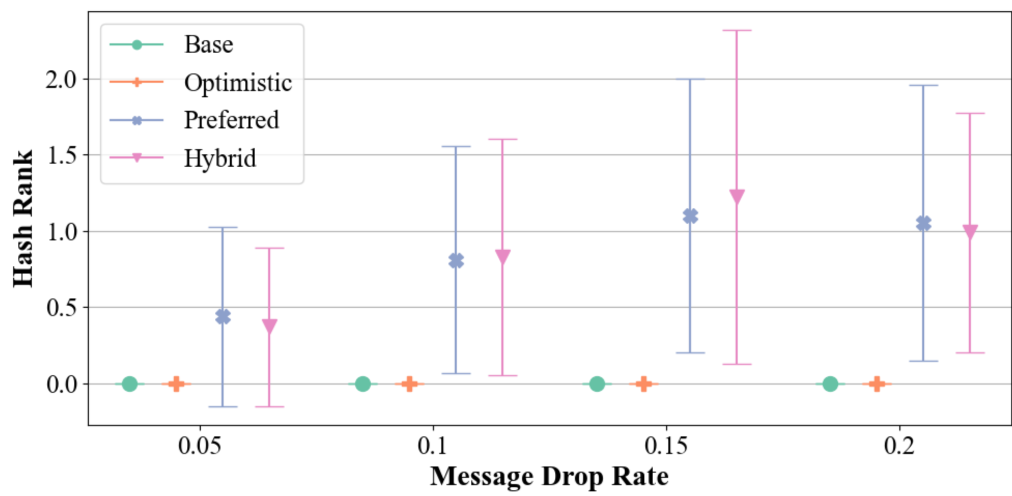


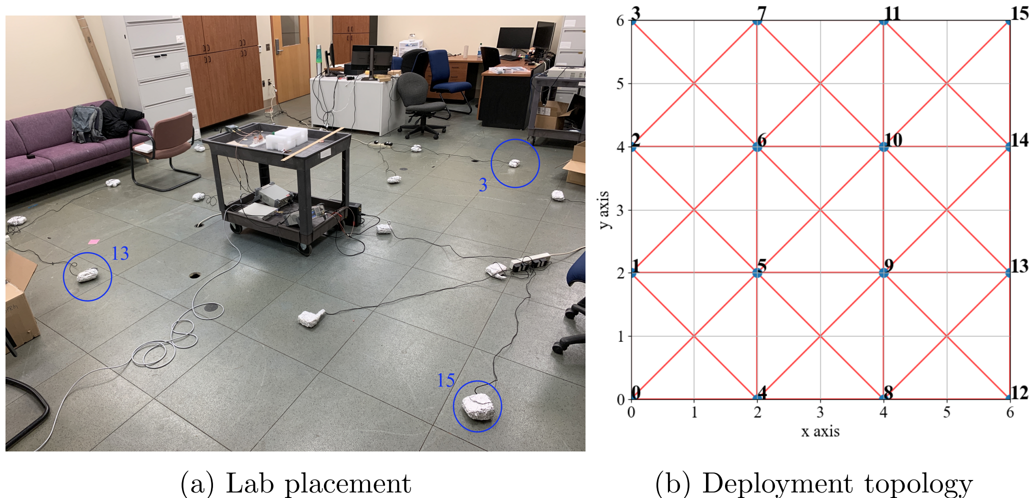
Figure 6.9: *Leader hash rank with different drop rates*

6.2 Deployment with Raspberry Pis

We implement our four c-tolerant leader election protocols for Raspberry Pi 4 devices, using about 2500 lines of Java code. We reuse code from the authors of the Medley system [13] as the membership layer (with their default configuration parameters) and layer our election protocols atop it. Separately, we also deployed Zookeeper [23] on the same Raspberry Pis, and compare it against our protocols.

6.2.1 Raspberry Pi Deployment Results

Deployments involve a default of 16 Raspberry Pi 4 devices connected in an ad-hoc mesh network. The topology is a 4x4 grid shown in Fig. 6.10b and the actual lab placement is shown in Fig. 6.10a. Each device is a Raspberry Pi 4 model B, with 2GB LPDDR4 RAM and Broadcom BCM2711, 1.5 GHz quad-core Cortex-A72 CPU. Based on recommendations [13], we attenuate the transmit power of each device to about 15 dBm. We use OLSRD for packet routing due to its easy configurability and popularity.



(a) Lab placement

(b) Deployment topology

Figure 6.10: *Topology of Raspberry Pi deployment*

The parameters and the network configurations are similar to those used in the simulation results of Sec. 6.1. The c values are set to $2, 3, 5, 7$ for message drop rates $0.05, 0.1, 0.15, 0.2$ respectively, based on our measurements from 5-minute runs of the failure detector [13].

Fig. 6.11 shows the completion time vs. message drop rates. Overall, the

comparative performance among the four protocols is similar to the simulation results (Fig. 6.5), thus validating that our simulator of Sec. 6.1 successfully models practically observed behavior. We do observe that the completion time (across protocols) varies a bit less in the Pi deployment than in the simulation. This is because the deployment topology is smaller (16 nodes) compared to the simulation (49 nodes)—lower system size results in lower c values and thus fewer messages.

Fig. 6.12 shows the number of leader change counts (during a given run) vs. message drop rates. Again, the comparative trends parallel simulation results (Fig. 6.7), with a couple of exceptions. We do observe that the leader change count for the Optimistic Protocol is relatively higher than in the simulation results. This is because the effective message drop rates in deployment are higher than the drop rates we set, e.g., Raspberry Pis themselves can drop packets even with message drop rate set to 0. This higher drop rate leads to more missing entries in membership lists and thus more leader changes. On the contrary, leader change count for the Hybrid Protocol is less than in simulation at high drop rates—this is because the initiator receives fewer RESPONSE (lower c) messages thus limiting the number of leader changes.

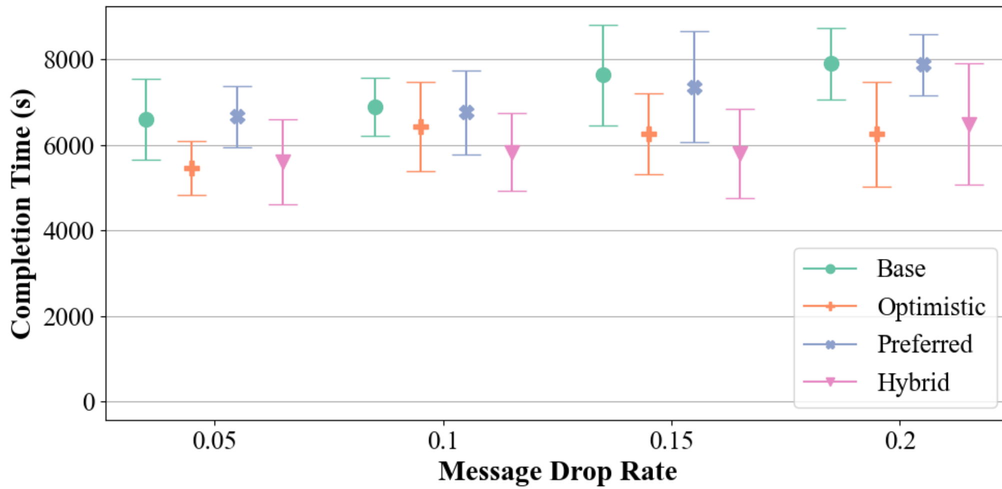


Figure 6.11: *Completion time with different drop rates on Raspberry Pis*

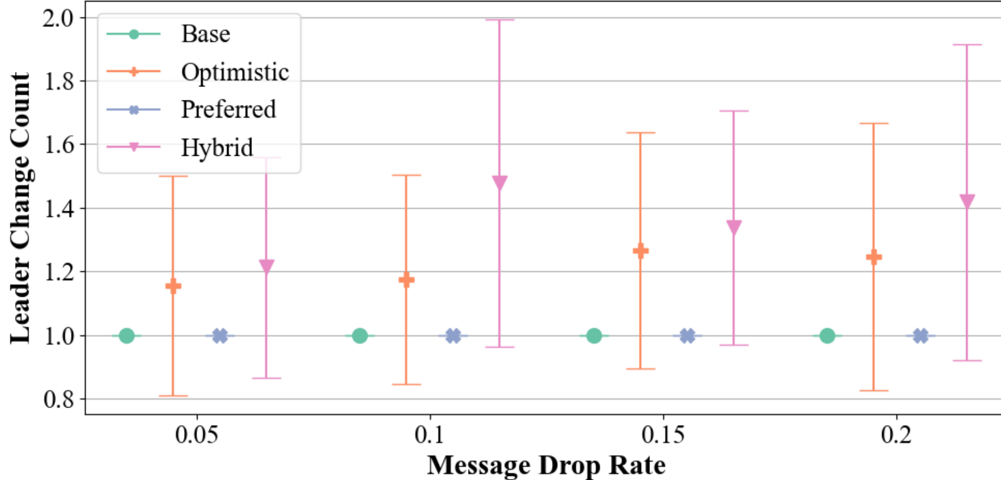


Figure 6.12: *Leader change count with different drop rates on Raspberry Pis*

6.2.2 Resource Utilization Comparison with Zookeeper

Carefully designed protocols for edge settings can offer significant benefits over “stock” open-source software. To illustrate this, we compare against a state-of-the-art coordination and election system. Specifically, we enabled Zookeeper [23] to run on the Raspberry Pi 4 devices, and we compare against our c-tolerant leader election protocol (our Hybrid Protocol) on the same settings as Sec. 6.1 (16 Raspberry Pis connected in a 4x4 grid mesh). Both Zookeeper and our c-tolerant election run continuously for around 10 minutes, and we record their memory usage, CPU usage, and network traffic (via tcpdump [68]). For Zookeeper, we set parameter values as suggested by the official documentation [69] ($tickTime = 2000, initLimit = 5, syncLmit = 2$).

Fig. 6.13 shows the memory utilization and Fig. 6.14 shows the CPU utilization (averaged across all nodes). In the stable state (after time $t = 100s$), our c-tolerant election utilizes 26.9% less CPU and 5.9% less memory than Zookeeper. During the warm-up phase ($t < 100s$), there is a spike in memory used by c-tolerant election when it runs the election, however even the spike uses less than 4% memory and it’s less than 13% above the stable memory usage.

Fig. 6.15 shows the total network bandwidth across all nodes during the 10-min run. The average bandwidth consumed by our c-tolerant election protocol is 20 Kbps, which is 82.9% less than Zookeeper’s average bandwidth (117 Kbps). Further, our c-tolerant election’s network usage is more stable

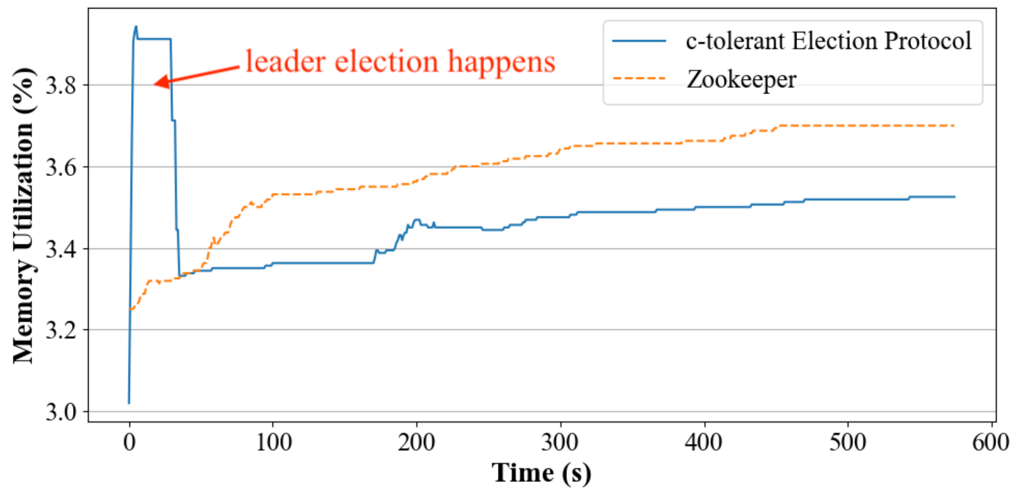


Figure 6.13: *Memory Utilization Comparison*

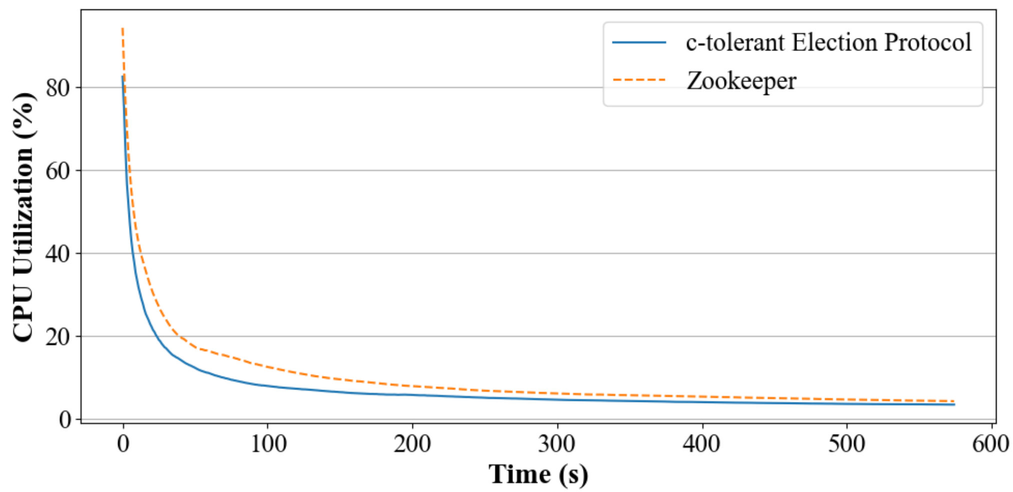


Figure 6.14: *CPU Utilization Comparison*

than Zookeeper, with a standard deviation over 10× lower: the respective standard deviations are 2.1 Kbps and 25.1 Kbps.

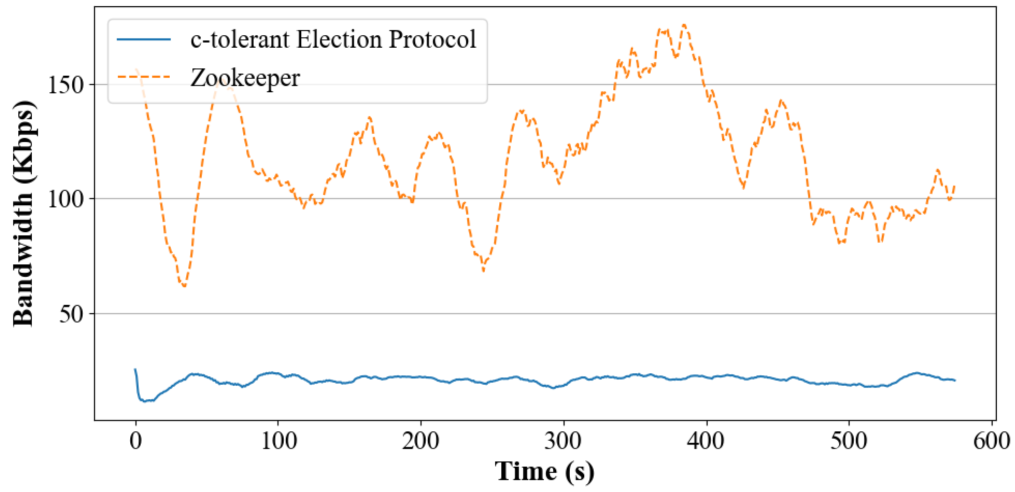


Figure 6.15: *Network Bandwidth Comparison*

CHAPTER 7

ESTIMATING CHURN PARAMETER c

In this chapter, we aim to estimate the value of c accurately as the underlying membership changes continuously. Sec. 7.1 presents the motivation. Sec. 7.2 presents the design of the two protocols: Z-score Sampling Protocol and Feedback Protocol. Sec. 7.3 presents the simulation of our protocols as well as the effect of underestimating churn c.

7.1 Motivation

Our churn-tolerant leader election protocols [18] are effective due to their ability to utilize an appropriate value of c for different network configurations. Underestimating c can lead to safety violations, where a node that does not have the lowest hash may be elected as the leader. On the other hand, overestimating c can result in unnecessary network bandwidth consumption, as the protocols will need to send extra Query and Response messages. For instance, Table 5.1 presents the c values for several network configurations running with Medley-F. However, these values may not be effective if some parameters change, or the leader election protocol operates on top of another membership service.

Our objective is to accurately estimate the value of c by employing two different protocols: the Z-score Sampling Protocol and the Feedback Protocol. The Z-score Sampling Protocol obtains a sample of membership lists from neighboring nodes and utilizes this information to estimate c. On the other hand, the Feedback Protocol estimates c based on the number of different proposed leaders during leader election. We have conducted simulations and experiments to demonstrate the effectiveness of both protocols and the consequences of underestimating c. Our aim is to estimate c dynamically as the underlying membership changes. It is assumed that an initial value of c is

set before the protocol commences.

7.2 Protocol Design

7.2.1 Z-score Sampling Protocol

The Z-score Sampling Protocol centers on estimating the value of c by analyzing membership lists from a small subset of nodes in the system. The protocol is based on the underlying assumption that if a particular node M_i is absent from the membership lists of other nodes, then the absence of M_i is uniformly distributed across all nodes in the system. This allows for a small sample of nodes to be representative of the entire network. While the leader node is typically chosen to initiate the protocol for simplicity, any node within the system can perform this function.

The Z-score Sampling Protocol is initiated by the leader node, who requests membership lists from a randomly-selected subset of s nodes. Each node responds with its full membership list, and the leader then identifies the node (denoted as M_i) that is absent from the most membership lists. Assuming that M_i is absent from k membership lists, the leader calculates an estimate of c using the formula $c = k \cdot N/s$, where N is the total number of nodes in the system. Once the estimate of c is calculated, the leader broadcasts this value to the entire network. The specific steps of the Z-score Sampling Protocol are described in Algorithm 6. If the membership of the network changes continuously, the leader can run the protocol periodically to maintain an accurate estimate of c.

An important consideration when implementing the Z-score Sampling Protocol is determining the appropriate sample size s. Suppose we aim to estimate c within a margin of error $\pm E$ (expressed as a percentage of the true value) with a confidence level of x, and let p be the probability that a node is missing from a given membership list. In this case, the minimum required sample size can be calculated using the following formula:

$$s = p \times (1 - p) \times (\frac{z}{E})^2$$

Algorithm 6 Z-score Sampling Protocol

```
1: function InitiateSample(s) // At Leader
2:     freqs ← empty dictionary
3:     Send SampleQuery to arbitrary s nodes
4:     for msg in received SampleResponse do
5:         membership ← msg
6:         for node in membership do
7:             if node not in freqs then
8:                 freqs[node] ← 0
9:             end if
10:            freqs[node] ← freqs[node] + 1
11:        end for
12:    end for
13:    k ← 0
14:    for node in freqs do
15:        k ← max(k, N − freqs[node])
16:    end for
17:    c ← k · N/s
18:    Broadcast c to all nodes
19: end function
20:
21: function RecvMessage(msg from node i) // Any Node
22:    if msg is of type SampleQuery then
23:        Send SampleResponse (← entire membership) to node i
24:    end if
25: end function
```

where

$$z = x + \frac{1-x}{2}$$

is the z-score for confidence level x. Based on Table 5.1, we find that p is at most 10% for most network configurations. Assuming $p = 0.1$, Table 7.1 provides a set of sample sizes s for different combinations of confidence levels x and margins of error E. For instance, if a system comprises 1000 nodes and we require the estimated c value to be within ± 20 (equivalent to 5% of 1000) of the actual c value with a 90% confidence interval, the sample size must be at least 32.

Table 7.1: *s values with different confidence level x and marge of error E*

E \ x	90%	95%
2%	203	214
5%	32	34
10%	8	9

One disadvantage of the Z-score Sampling Protocol is that to achieve high accuracy with high confidence, the required sample size can increase significantly. Furthermore, the protocol does not leverage estimation results obtained from multiple protocol runs, which can be wasteful if the system needs to dynamically estimate c by running the Z-score Sampling Protocol multiple times. To address these issues, we introduce the Feedback Protocol, which will be discussed in the following subsection.

7.2.2 Feedback Protocol

The aim of the Feedback Protocol is to refine the estimate of c based on intermediate results obtained during the leader election process (via the Base protocol and Optimistic protocol). The fundamental idea is that if the initiator receives Response messages containing different candidates, the current c value may be underestimated. Conversely, if the Response messages contain only a single candidate, this suggests that the current c value is accurate or potentially overestimated.

Once the leader election process is complete, each initiator sends a message

to the leader indicating whether they received multiple candidates. Specifically, an initiator sends the value 1 if they received only one candidate, and 2 otherwise. The leader then computes the average of all the received values and assigns it to the variable d. Assuming the actual c value of the system is denoted as c_1, and the current c value estimated by the protocol is denoted as c_2, we can represent d as follows:

$$\begin{aligned} d &= (1-p)\cdot 2 + p\cdot 1 \\ &= (1-(1-\frac{c_1}{N})^{c_2+1})\cdot 2 + (1-\frac{c_1}{N})^{c_2+1}\cdot 1 \\ &= 2-(1-\frac{c_1}{N})^{c_2+1} \end{aligned}$$

where

$$p = (1-\frac{c_1}{N})^{c_2+1}$$

is the probability that a node is not missing in any of the (c_2+1) RESPONSE messages received by the initiator. By solving the equation, we get

$$c_1 = N\cdot(1-exp(\frac{ln(2-d)}{c_2+1}))$$

Since the result from a single leader election run may be skewed, we set the c_2 value with an exponentially-weighted moving average:

$$c_2 = \alpha\cdot c_1 + (1-\alpha)\cdot c_2 \tag{7.1}$$

We set $\alpha = 0.1$ for our protocol. The detailed protocol description is shown in Algorithm 7.

The Feedback Protocol has a limitation in that each estimation necessitates a leader election run. If the system conducts frequent leader elections, the estimated c value can converge to the actual c value relatively quickly. However, if leader elections are infrequent, it may take a considerable amount of time to obtain an accurate estimation of c.

To explore the benefits of both the Z-score Sampling Protocol and the Feedback Protocol, we implemented the exponentially-weighted moving average into our Z-score Sampling Protocol, which we call the Sample-Window Protocol. We use c_1 to represent the result from the Z-score Sampling Protocol and c_2 as the current estimate of c. To estimate the new value of c,

Algorithm 7 Feedback Protocol

```
 1: function InitiateFeedback(c_2) // At Leader
 2:     count ← 0, total ← 0
 3:     for msg in received FeedbackValue do
 4:         val ← msg
 5:         count ← count + 1, total ← total + val
 6:     end for
 7:     d = total / count
 8:     c = N · (1 − exp(ln(2−d) / (c_2+1)))
 9:     c_2 = α · c + (1 − α) · c_2
10:     Broadcast c_2 to all nodes
11: end function
12:
13: function InitiateFeedback // Any Initiator
14:     if Only receive a single candidate from all Response messages then
15:         val ← 1
16:     else
17:         val ← 2
18:     end if
19:     Send val to leader
20: end function
```

we apply Eqn. 7.1. To obtain an accurate estimation of c, the system can execute the hybrid protocol multiple times.

7.3 Simulation

We would like to address the following questions through our simulation:

1. What is the effect of using underestimated c for our churn-tolerant election protocol? And how does the election result change if multiple initiators exist?

2. How accurate is the c estimation for our Z-score Sampling Protocol and Feedback Protocol?

3. For the Feedback Protocol, how quickly does the estimated c converge to the actual c value?

To simulate our churn-tolerant election protocols, we developed a custom simulator using Java. To control the actual value of c, we generated the

underlying membership at each node differently from the trace in Medley-F. Specifically, we generated memberships such that nodes with low hashes were absent from exactly c memberships of other nodes, as predetermined. The data points presented in the figures were collected from 100 independent simulation runs.

7.3.1 Effect of Underestimating Churn

To demonstrate the significance of selecting the appropriate c, we executed the leader election protocol Base Protocol with a lower c value than the actual c value for the specified network configurations. The system comprised of 128 nodes, and we generate the membership such that node 0 is absent from the membership lists of exactly 25 arbitrary nodes. During the simulation, each leader election round involved 5 initiators.

Fig. 7.1 shows the average hash rank of the elected leader with different c values, where all of them are less than the actual c value for the system. As expected, when c is low, the leader hash rank is high because some initiators do not receive adequate responses to determine the correct leader. However, as c increases, the hash rank decreases, eventually reaching 0, which implies that the protocol meets the safety property.

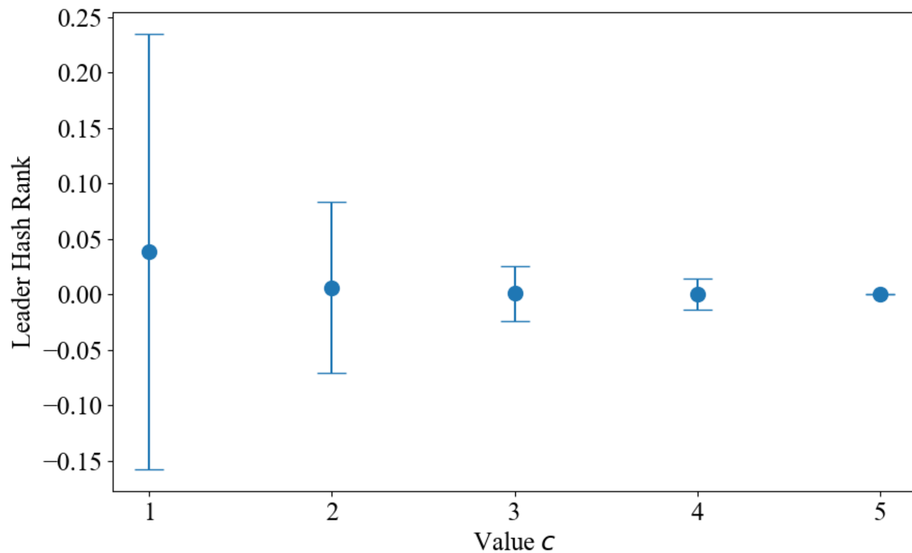


Figure 7.1: *Base Protocol hash rank with different c*

Fig. 7.2 shows the average hash rank of the elected leader with different numbers of initiators, with the fixed value of $c = 3$ which is less than the actual c value. As the number of initiators increases, hash rank increases only slightly, implying that the number of initiators does not affect safety too much.

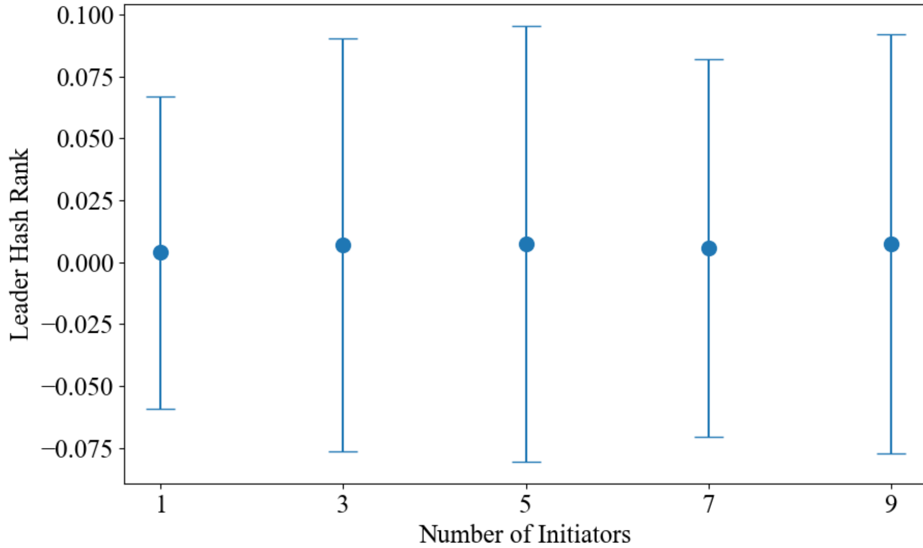


Figure 7.2: *Base Protocol hash rank with different number of initiators*

In summary, the probability of selecting the incorrect leader increases as c decreases or the number of initiators increases. However, even in the worst-case scenario, the hash rank remains small. Nonetheless, the protocol must still choose the appropriate c to satisfy the safety property.

7.3.2 Protocol Simulation

We simulate both the Z-Score Sampling Protocol and the Feedback Protocol. During simulation, the underlying membership is generated continuously with moving churn rates (different c values). The c value is randomly chosen with mean of $\frac{N}{10}$ and standard deviation of $\frac{N}{20}$, where N is the total number of nodes in the system.

Fig. 7.3 shows the difference between the estimated c and the actual c value for the Feedback Protocol, Z-Score Sampling Protocol, and Sample-Window protocol with different sample sizes. As the sample size increases,

the estimation becomes more precise, but at the cost of additional network bandwidth usage due to extra messages. The Feedback Protocol exhibits a smaller standard deviation and more accurate estimation than the Z-Score Sampling Protocol at smaller node numbers. However, it has a tendency to underestimate the actual c value. Furthermore, the Feedback Protocol is only executed during leader election, whereas the Z-Score Sampling Protocol can be executed without restriction. On the other hand, the Sample-Window Protocol has significantly lower standard deviation than the Z-Score Sampling Protocol, but the estimated value is less accurate on average.

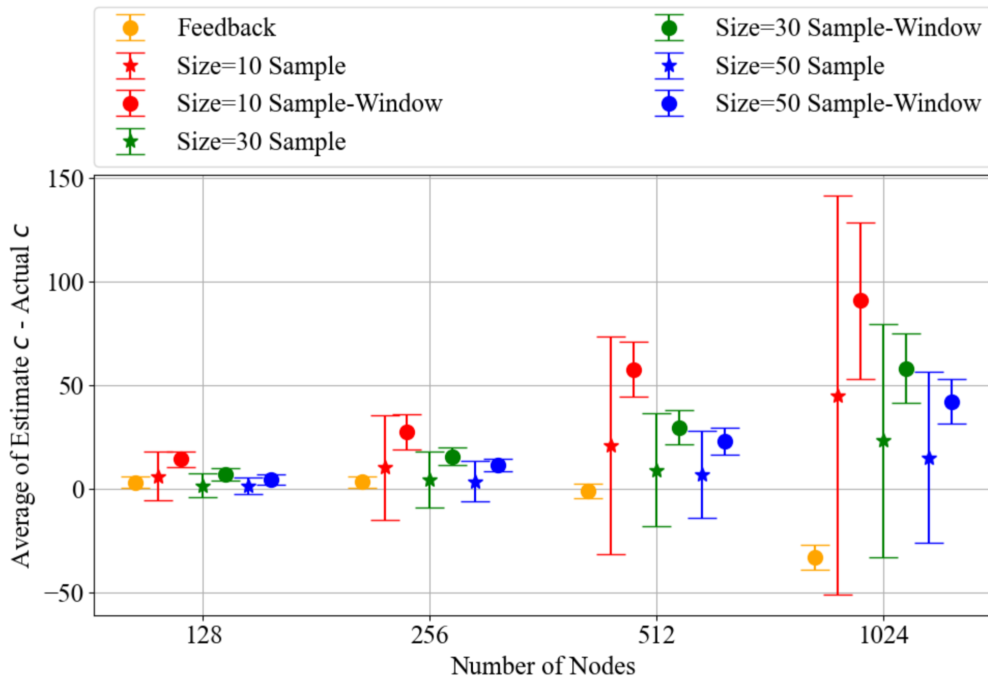


Figure 7.3: *Difference between estimated and actual value of c*

Fig. 7.4 shows an example of Feedback Protocol run with 100 rounds of leader election. There are 128 nodes in the system and we set $\alpha = 0.2$. We can see that the estimated value matches closely to the actual value of c during most of the time.

In the previous example, the estimated starting value of c was close to the actual value. However, what if we do not know the actual value at the beginning? To determine how quickly the Feedback Protocol converges from an arbitrary starting point, we fixed the underlying membership and initiated the Feedback Protocol with $c = 1$. The simulation consists of 128 nodes, and

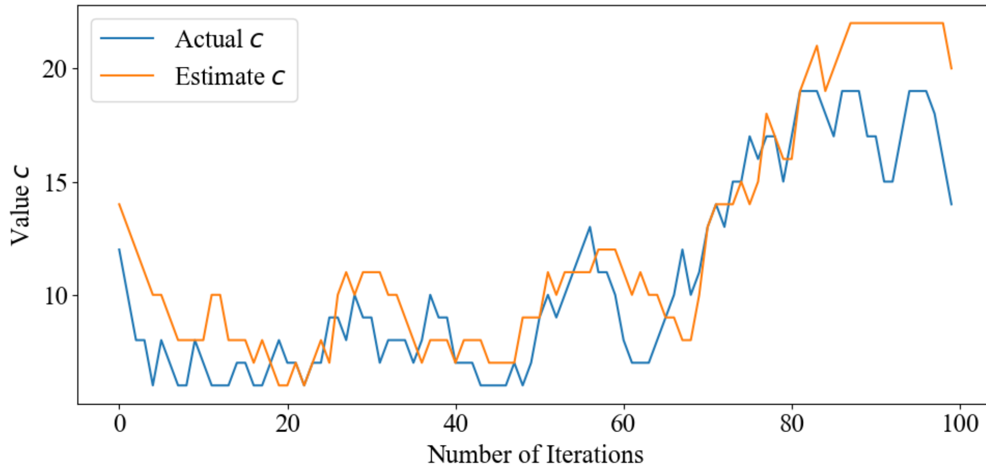


Figure 7.4: *Example of the Feedback Protocol experiment run*

the actual value of c is set to 13. Fig. 7.5 illustrates the distribution of convergence time (number of election rounds) against the difference between the estimated and actual values of c. The figure indicates that the estimate becomes more accurate with additional election rounds and that it takes roughly 20 rounds for the Feedback Protocol to converge to the actual value of c.

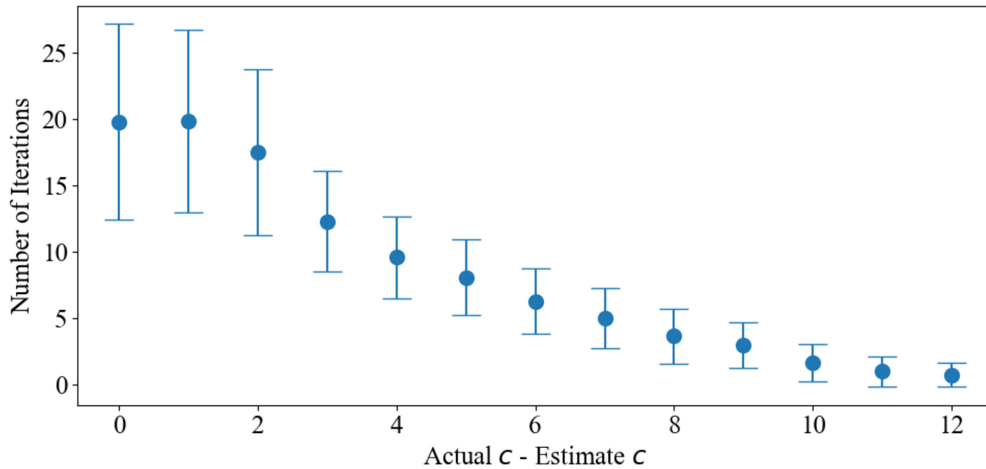


Figure 7.5: *Number of election rounds v.s. difference between estimated and actual value of c*

7.4 Conclusion

We conclude that the Feedback Protocol has the smallest standard deviation, and the Z-score Sampling Protocol has the largest standard deviation. The Feedback Protocol estimates accurately with small topologies, but it tends to underestimate with large topologies. On the other hand, both the Z-score Sampling Protocol and the Sample-Window Protocol tend to overestimate, especially at large topologies. We recommend that:

- If the system has regular leader election events, the Feedback Protocol is preferred. Otherwise, the Z-score Sampling Protocol and the Sample-Window Protocol are preferred.
- The Feedback Protocol is preferred at small (≤ 512 nodes) topologies.
- The Sample-Window Protocol is preferred at large topologies, but the result of the estimation needs to be adjusted "down" carefully.

CHAPTER 8

CONCLUSION

8.1 Summary

We summarize our contributions as follows:

- We presented Medley-F, a decentralized membership service for distributed IoT systems that operate on wireless ad-hoc networks. Medley-F can identify failures as quickly as the traditional SWIM method while reducing the product of failure detection time and communication cost by 37.8%. By using active and passive feedback, we can lower tail detection time by up to 31% and dissemination time by up to 54%.
- We presented a family of four churn-tolerant (c-tolerant) leader election protocols intended for edge environments that face churn. Our protocols only require a weakly-consistent membership protocol running underneath it. Our four protocols satisfy Safety and Liveness, use provably minimal messages, and elect healthy leaders with high probability. Our experiments with both trace-driven simulations, as well as a Raspberry Pi deployment, showed that: (i) our Optimistic Protocol reduces leader election completion time by 42.6% (vs. Base Protocol), (ii) our Preferred Protocol elects healthier leaders, and (iii) compared to "stock" Zookeeper, our c-tolerant election's bandwidth usage is 82.9% less and 10× more stable, and its memory and CPU usage are respectively 5.9% and 26.9% lower.
- We presented two protocols that estimate churn dynamically with continuous membership changes. Our Z-Score Sample Protocol utilizes information from a small subset of memberships to estimate churn, and our feedback protocol utilizes intermediate results during leader election to estimate churn. We also presented the Sample-Window Protocol that combines the idea of both protocols. Our simulation shows that all proto-

cols (with sample size = 50) can estimate churn within 4.5% of accuracy on average. Our simulation also shows that even with underestimated churn value, safety property is only violated with 4% of all election runs in the worst case.

8.2 Future Work

We discuss three major future directions:

- One possible direction is to design additional distributed protocols that can tolerate churn. These may include protocols for mutual exclusion, task scheduling, consensus, and distributed hash tables. Most current research is based on the assumption of complete and accurate membership knowledge and only focuses on tolerating failures rather than missing entries.
- Another possible direction is in leader election for systems with high churn rates. Our churn-tolerant election protocols are most effective when the churn rate is low. However, in high-churn scenarios, our protocol may require potentially $O(N)$ messages, where N is the size of the system. Examples of high-churn systems include distributed file-sharing systems such BitTorrent, Distributed web crawling, and Blockchain networks. The churn rate is high since each node can join and leave at any time, and the system should be adapted to these changes. In these situations, where bandwidth usage is already high due to churn, it is important to minimize additional bandwidth incurred. Possible solutions include designing election protocols that can operate with high churn rates while using less bandwidth, even if they take longer to complete. Additionally, high churn rates may require protocols that can tolerate frequent leader failures, which may require further optimization.
- Future research could explore applications that leverage leader election to achieve agreement among nodes in distributed systems. In smart farms, leader election can enable sensors to coordinate actions based on their readings, such as when to water crops or apply fertilizers. Similarly, leader election can be used in satellite constellations to select a leader satellite that can coordinate actions, such as adjusting orbits or coordinating data

transmission. Robust leader election protocols that can handle failures and adapt to changes in network topology will be critical to building such applications.

Other minor optimizations that are feasible on our work:

- For ease of usage, one may combine our churn-tolerant election protocols and a weakly-consistent membership protocol into a single application. It can have functionalities and APIs similar to widely-used software like Zookeeper.
- Current election protocols do not have any mechanism if safety is violated due to churn underestimation. Ideally, the system should be able to recover from any incorrect state, i.e., multiple leaders are elected.
- Our sample protocol for churn estimation assumes that the given node is uniformly missing in other memberships and relies on this uniformity for its algorithm. However, this assumption may not hold as false positives tend to cluster in certain regions. For instance, if node x incorrectly identifies node y as failed, it is more likely that the nodes around node x will also incorrectly mark node y as failed. To address this, one potential optimization is to utilize topology-aware sampling and a more advanced algorithm to estimate churn accurately.

REFERENCES

[1] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[2] "Global edge computing market to reach $156 billion by 2030," 2022. [Online]. Available: https://www.techrepublic.com/article/global-edge-computing-market/

[3] "Internet of Things: Key stats for 2022," 2022. [Online]. Available: https://techinformed.com/internet-of-things-key-stats-for-2022/

[4] "EarthSense TerraSentia robots," 2022. [Online]. Available: https://www.earthsense.co

[5] "EarthSense cover crop robots," 2022. [Online]. Available: https://www.earthsense.co/regen

[6] A. N. Sivakumar, S. Modi, M. V. Gasparino, C. Ellis, A. E. B. Velasquez, G. Chowdhary, and S. Gupta, "Learned visual navigation for under-canopy agricultural robots," *arXiv preprint arXiv:2107.02792*, 2021.

[7] "Planet Inc. dove satellite constellation," 2022. [Online]. Available: https://www.planet.com/our-constellations/

[8] "Starlink satellite constellation," 2022. [Online]. Available: https://www.starlink.com/

[9] M. A. Khan, H. E. Sayed, S. Malik, T. Zia, J. Khan, N. Alkaabi, and H. Ignatious, "Level-5 autonomous driving—are we there yet? a review of research literature," *ACM Computing Surveys (CSUR)*, vol. 55, no. 2, pp. 1–38, 2022.

[10] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi, and A. Rindos, "Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric)," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2017, pp. 253–255.

[11] "Java Transaction API," 2022. [Online]. Available: https://www.oracle.com/java/technologies/jta.html

[12] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin et al., “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[13] R. Yang, J. Wang, J. Hu, S. Zhu, Y. Li, and I. Gupta, “Medley: A membership service for iot networks,” *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2492–2505, 2022.

[14] K. Kapitanova, E. Hoque, J. A. Stankovic, K. Whitehouse, and S. H. Son, “Being smart about failures: assessing repairs in smart homes,” in *Proceedings of ACM UbiComp*, 2012, pp. 51–60.

[15] J. Ye, G. Stevenson, and S. Dobson, “Detecting abnormal events on binary sensors in smart home environments,” *Pervasive and Mobile Computing*, vol. 33, pp. 32–49, 2016.

[16] A. K. Sikder, H. Aksu, and A. S. Uluagac, “6thsense: A context-aware sensor-based attack detector for smart devices,” in *Proceedings of USENIX Security*, 2017, pp. 397–414.

[17] J. Choi, H. Jeoung, J. Kim, Y. Ko, W. Jung, H. Kim, and J. Kim, “Detecting and identifying faulty IoT devices in smart home with context extraction,” in *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 610–621.

[18] J. Wang and I. Gupta, “Churn-tolerant leader election protocols,” in *Proceedings 43rd International Conference on Distributed Computing Systems*. IEEE, in press.

[19] D. Stutzbach and R. Rejaie, “Understanding churn in peer-to-peer networks,” in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006, pp. 189–202.

[20] A. Binzenhöfer and K. Leibnitz, “Estimating churn in structured p2p networks,” in *Managing Traffic Performance in Converged Networks: 20th International Teletraffic Congress, ITC20 2007, Ottawa, Canada, June 17-21, 2007. Proceedings.* Springer, 2007, pp. 630–641.

[21] P. B. Godfrey, S. Shenker, and I. Stoica, “Minimizing churn in distributed systems,” *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 147–158, 2006.

[22] R. Yang, S. Zhu, Y. Li, and I. Gupta, “Medley: A novel distributed failure detector for iot networks,” in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 319–331.

[23] "Apache Zookeeper," 2022. [Online]. Available: https://zookeeper.apache.org/

[24] M. Kawazoe Aguilera, W. Chen, and S. Toueg, "Heartbeat: A timeout-free failure detector for quiescent reliable communication," in *Distributed Algorithms: 11th International Workshop, WDAG'97 Saarbrücken, Germany, September 24–26, 1997 Proceedings 11*. Springer, 1997, pp. 126–140.

[25] R. Van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Middleware'98*. Springer, 1998, pp. 55–70.

[26] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 system architecture," *IBM Systems Journal*, vol. 34, no. 2, pp. 414–446, 1995.

[27] I. Gupta, T. D. Chandra, and G. S. Goldszmidt, "On scalable and efficient distributed failure detectors," in *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, 2001, pp. 170–179.

[28] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable weakly-consistent infection-style process group membership protocol," in *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2002, pp. 303–312.

[29] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[30] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[31] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the 11th ACM Symposium on Operating systems principles*, 1987, pp. 123–138.

[32] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319.

[33] Y. Amir, D. Dolev, S. Kramer, and D. Malki, *Transis: A communication sub-system for high availability*. Hebrew University of Jerusalem. Leibniz Center for Research in Computer Science, 1991.

[34] O. Babaoglu, R. Davoli, L.-A. Giachini, and M. G. Baker, "RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems," in *Proceedings of 28th HICSS*, vol. 2. IEEE, 1995, pp. 612–621.

[35] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[36] Z. J. Haas, J. Y. Halpern, and L. Li, "Gossip-based ad hoc routing," *IEEE/ACM Transactions on Networking*, vol. 14, no. 3, pp. 479–491, 2006.

[37] K. Birman, "The promise, and limitations, of gossip protocols," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 8–13, 2007.

[38] I. Gupta, K. P. Birman, and R. Van Renesse, "Fighting fire with fire: using randomized gossip to combat stochastic scalability limits," *Quality and Reliability Engineering International*, vol. 18, no. 3, pp. 165–184, 2002.

[39] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz et al., "Handling churn in a dht," in *Proceedings of the USENIX Annual Technical Conference*, vol. 6. Boston, MA, USA, 2004, pp. 127–140.

[40] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh, "Efficient epidemic-style protocols for reliable and scalable multicast," in *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* IEEE, 2002, pp. 180–189.

[41] J. Dunagan, N. J. Harvey, M. B. Jones, D. Kostic, M. Theimer, and A. Wolman, "FUSE: Lightweight guaranteed distributed failure notification," in *Proceedings of USENIX OSDI*, 2004.

[42] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, 2005, pp. 255–267.

[43] S. Rost and H. Balakrishnan, "Memento: A health monitoring system for wireless sensor networks," in *2006 3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks*, vol. 2. IEEE, 2006, pp. 575–584.

[44] B.-r. Chen, G. Peterson, G. Mainland, and M. Welsh, "Livenet: Using passive monitoring to reconstruct sensor network dynamics," in *Distributed Computing in Sensor Systems: 4th IEEE International Conference, DCOSS 2008 Santorini Island, Greece, June 11-14, 2008 Proceedings 4*. Springer, 2008, pp. 79–98.

[45] R. N. Duche and N. P. Sarwade, "Sensor node failure detection based on round trip delay and paths in WSNs," *IEEE Sensors Journal*, vol. 14, no. 2, pp. 455–464, 2014.

[46] M. Asim, H. Mokhtar, and M. Merabti, "A fault management architecture for wireless sensor network," in *2008 International Wireless Communications and Mobile Computing Conference*. IEEE, 2008, pp. 779–785.

[47] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–37, 2018.

[48] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi et al., "Service fabric: a distributed platform for building microservices in the cloud," in *Proceedings of the 13th EuroSys Conference*, 2018, pp. 1–15.

[49] I. Gupta, R. Van Renesse, and K. P. Birman, "A probabilistically correct leader election protocol for large groups," in *International Symposium on Distributed Computing*. Springer, 2000, pp. 89–103.

[50] I. Gupta, R. Van Renesse, and K. P. Birman, "Scalable fault-tolerant aggregation in large process groups," in *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2001, pp. 433–442.

[51] C. Sengul, M. J. Miller, and I. Gupta, "Adaptive probability-based broadcast forwarding in energy-saving sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 4, no. 2, pp. 1–32, 2008.

[52] F. Kuhn, S. Schmid, and R. Wattenhofer, "A self-repairing peer-to-peer system resilient to dynamic adversarial churn," in *International Workshop on Peer-to-Peer Systems*. Springer, 2005, pp. 13–23.

[53] S. Legtchenko, S. Monnet, P. Sens, and G. Muller, "Relaxdht: A churn-resilient replication strategy for peer-to-peer distributed hash-tables," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 2, pp. 1–18, 2012.

[54] H. Garcia-Molina, “Elections in a distributed computing system,” *IEEE Transactions on Computers*, vol. 31, no. 01, pp. 48–59, 1982.

[55] L. Lamport, “The part-time parliament,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[56] N. Malpani, J. L. Welch, and N. Vaidya, “Leader election algorithms for mobile ad hoc networks,” in *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, 2000, pp. 96–103.

[57] S. Vasudevan, J. Kurose, and D. Towsley, “Design and analysis of a leader election algorithm for mobile ad hoc networks,” in *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.* IEEE, 2004, pp. 350–360.

[58] S. Dolev, A. Israeli, and S. Moran, “Uniform dynamic self-stabilizing leader election,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 4, pp. 424–440, 1997.

[59] M. Fischer and H. Jiang, “Self-stabilizing leader election in networks of finite-state anonymous agents,” in *International Conference on Principles of Distributed Systems.* Springer, 2006, pp. 395–409.

[60] S.-T. Huang, “Leader election in uniform rings,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 3, pp. 563–573, 1993.

[61] R. Chandra, V. Ramasubramanian, and K. Birman, “Anonymous gossip: Improving multicast reliability in mobile ad-hoc networks,” in *Proceedings 21st International Conference on Distributed Computing Systems.* IEEE, 2001, pp. 275–283.

[62] J. C. Lin and S. Paul, “RMTP: A reliable multicast transport protocol,” in *Proceedings of IEEE INFOCOM’96. Conference on Computer Communications*, vol. 3. IEEE, 1996, pp. 1414–1424.

[63] “Raspberry Pi 4 model B,” 2016. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[64] “Optimized link state routing protocol.” [Online]. Available: https://tinyurl.com/olsrd-wiki

[65] D. Mazieres, “The stellar consensus protocol: A federated model for internet-level consensus,” *Stellar Development Foundation*, vol. 32, pp. 1–45, 2015.

[66] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich et al., "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the 13th EuroSys Conference*, 2018, pp. 1–15.

[67] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco, "Single secret leader election," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 12–24.

[68] "Tcpdump," 2022. [Online]. Available: https://www.tcpdump.org

[69] "Apache Zookeeper guide," 2022. [Online]. Available: https://zookeeper.apache.org/doc/r3.3.3/zookeeperStarted.html